Semelhante às liguagens C, C++ e Pascal, a linguagem Java também dá suporte a vetores e matrizes (*arrays*) de diversas formas. Os vetores constituem uma forma muito conveniente de organizar informações em fileira. Por exemplo, podemos formar um vetor com as notas de cinco alunos de uma sala de aula do seguinte modo:

float nota[] = { 7.8, 8.4, 4.2, 1.8, 6.4 };

Neste caso **nota[0]** é a nota do primeiro aluno, isto é, 7.8, **nota[1]** é a nota do segundo, ou seja, 8.4, e assim por diante.

A utilização de vetores e matrizes em Java envolve três etapas:

1. **Declarar o vetor ou matriz.** Para isto, basta acrescentar um par de colchetes antes ou depois do nome da variável. Por exemplo:
2. int ind[];
3. double A[][],T[][][];

int []nota;

1. **Reservar espaço de memória e definir o tamanho**. É preciso definir o tamanho do vetor, isto é, a quantidade total de elementos que terá de armazenar. Em seguida é necessário reservar espaço de memória para armazenar os elementos. Isto é feito de maneira simples pelo operador **new**:
2. ind = new int[10];
3. nota = new int[70];

A = new double[10][20];

1. **Armazenar elementos no vetor ou matriz**. Para armazenar uma informação em um dos elementos de um vetor ou matriz, é necessário fornecer um índice que indique a posição desse elemento. Por exemplo, para armazenar um valor na quarta posição do vetor **nota**, fazemos o seguinte:

nota[3] = 5.2;

Como podemos observar, os índices começam em zero e vão até o número de posições reservadas, menos um. No vetor **nota** criado acima, os índices válidos vão de 0 até 69. Caso haja a tentativa de atribuir um valor a um elemento cujo índice esteja fora desse intervalo, ocorrerá um erro que impedirá a execução do programa. Por isso, é necessário um certo cuidado ao manejar com esses índices, garantindo o perfeito funcionamento do programa.

Existe, porém, um atalho para esses três passos quando desejamos criar um vetor com valores atribuídos de modo estático. Foi o que fizemos no primeiro exemplo acima, declarando o vetor **nota** com as notas de cinco alunos. Nesse caso, espaço suficiente para as notas de cinco alunos foi reservado e as notas foram guardadas em respectivas posições do vetor.

Entretanto, nem sempre é tão fácil assim. Em geral, estaremos interessados em trabalhar com vetores muito muito maiores, e cujos elementos sejam provenientes de outras fontes, que variam com o tempo. Assim, seremos obrigados a seguir os passos acima.

Eis mais alguns exemplos de vetores e matrizes:

// 12 primeiros termos da seqüência de Fibonacci:

long Fibonacci[] = {1,1,2,3,5,8,13,34,55,89,144};

// Tabela de sen(n\*pi/6), n=0,1,2,...5

float seno[] = {0.0000,0.5000,0.8660,1.0000,0.8660,0.5000};

// Tabela de log(1+n), n=0,2...99:

double tlog[] = new double[100];

for(int n=0; n<100; n++) tlog[i] = Math.log(1+n);

// Matriz dos coeficientes

double A[][] = { {1,2,3}, {0,1,3}, {0,0,-1} };

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part23.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), [seguinte](http://www.dm.ufscar.br/~waldeck/curso/java/part25.html)]

Em contraste com a estática dos dados, os *métodos* definem as ações a serem tomadas em diversos momentos da execução de um programa. Como em outras linguagens, como C, C++, Pascal, Fortran, etc, os métodos correspondem aos conceitos comuns de funções, procedimentos ou subrotinas. Estes são apenas conjuntos ordenados de declarações de dados, comandos e expressões. Em termos simples, são os métodos que realizam todas as tarefas para as quais o programa foi escrito, por exemplo, realizar cálculos, resumir informações de um arquivo, produzir um relatório, criar um gráfico, gerar um filme de animação, etc.

**Classes**

Os métodos, assim como os dados, têm um local de residência, as classes. Mais adiante, vamos estudar as [classes](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html) em detalhes. Por hora, precisamos apenas de alguns poucos conceitos para poder entender os métodos. Pensemos uma classe como sendo um conjunto de dados (variáveis) e métodos (funções) da forma:

class [nome] {

[dados e métodos]

}

onde **[nome]** é um [identificador](http://www.dm.ufscar.br/~waldeck/curso/java/part22.html#identificador) que define o nome da classe, e o par de chaves delimita uma região para declaração de variáveis e métodos. A declaração de variáveis já foi vista anteriormente no capítulo sobre [tipos de dados](http://www.dm.ufscar.br/~waldeck/curso/java/part22.html). Uma classe pode ser *privativa* ou *pública*. Uma classe privativa é declarada como no exemplo acima e é conhecida apenas no escopo delimitado pelo arquivo que a contém. Como um programa Java pode ser quebrado em múltiplos arquivos de código fonte distintos, pode ser necessário que as diversas partes integrantes do programa interajam, trocando dados entre si e chamando métodos umas das outras. Isso torna-se possível através das classes públicas, as quais são conhecidas por qualquer arquivo fonte que componha o programa. Para tornar uma classe pública, basta preceder sua declaração pela palavra-chave **public** como no seguinte exemplo:

public class [nome da classe] {

[dados e métodos]

}

Há uma convenção em Java que estabelece que deve haver exatamente uma classe pública para cada arquivo-fonte de que consiste um programa Java, e seu nome deve ser precisamente o nome do arquivo, sem o sufixo **.java**. Desse modo, existe uma correspondência biunívoca entre as classes públicas e os arquivos-fonte que as contém.

Podemos declarar uma classe a partir do chão, com todos os seus dados e métodos, ou podemos declarar uma classe derivando-a a partir de uma outra já existente. Este é um recurso muito útil, pois permite aproveitar um esforço de elaboração anterior, aumentando significativamente a produtividade da programação, e diminuindo o tamanho do código objeto. Suponhamos por exemplo, que tenhamos declarado previamente a seguinte classe:

class Polígono {

int cx, cy; // Coordenadas do centro do polígono

}

Esta classe define em linhas gerais o que é um polígono, guardando uma única característica comum a qualquer polígono, isto é, as coordenadas de seu centro. Agora, suponhamos que desejamos criar uma classe para guardar informações sobre um quadrado. Neste caso, não precisamos criar uma classe que dê as coordenadas do centro do quadrado assim como as suas dimensões. Basta fazer simplesmente:

class Quadrado extends Polígono {

int lado; // Comprimento de um lado do quadrado

}

A classe quadrado declarada desse modo se diz uma *classe derivada* da classe Poligono, da qual herda os dados (e os métodos) nela contidos. Esta declaração é equivalente a

class Quadrado {

int cx, cy; // Coordenadas do centro do polígono

int lado; // Comprimento de um lado do quadrado

}

Desejando fazer uma classe para representar um retângulo, bastaria fazer então

class Retângulo extends Polígono {

int base, alt; // Base e altura do retângulo

}

**Objetos**

Uma particular instância de uma classe é chamada *objeto*. Para entender a diferença entre classes e objetos, fazemos alusão à metáfora da fábrica de torradeiras. A fábrica de torradeiras não é uma torradeira, mas define o tipo de produto que sai dela, isto é, as torradeiras. Do mesmo modo a torradeira não é a fábrica, mas o produto feito por ela. Comparamos as classes às fabricas e os objetos aos produtos feitos por elas. Grosso modo, podemos dizer que as classes não ocupam espaço na memória, por serem abstrações, equanto que, os objetos ocupam espaço de memória por serem concretizações dessas abstrações.

Nas declarações acima, introduzimos algumas classes que permitem representar polígonos. Porém, não instanciamos nenhuma das classes criando particulares objetos a partir dessas classes. Por exemplo, a partir da classe quadrado, podemos fazer objetos representando quadrados de diversos comprimentos laterais, ou retângulos de diferentes dimensões:

Quadrado A, B, C;

Retângulo D;

A.lado = 1; // O quadrado A terá os lados de comprimento 1

B.lado = 2; // O quadrado B terá os lados de comprimento 2

C.lado = 7; // O quadrado C terá os lados de comprimento 7

D.base = 3; // O retangulo D terá base 3 e ...

D.alt = 4; // altura 4, e centrado na origem

D.cx = 0;

D.cy = 0;

As declarações acima são semelhantes às que vimos no capítulo anterior, com excessão de que no lugar do nome que identifica o tipo de dado estamos usando o nome de uma classe. Neste exemplo, as classes **Quadrado** e **Retângulo** foram empregadas para declarar os objetos (ou variáveis) A, B, C e D.

Em certo sentido as classes complementam os tipos de dados nativos da linguagem Java, com tipos de dados complexos criados pelo programador. Esse fato, aliado à possibilidade de derivar classes, tornam as linguagens orientadas a objetos extremamente producentes.

**Chamando métodos**

Um método entra em ação no momento em que é chamado. Isto pode ocorrer explicitamente ou implicitamente. A chamada explícita se dá por ordem do programador através da execução de um comando ou expressão contendo o nome do método. Em nosso programa [AloPessoal](http://www.dm.ufscar.br/~waldeck/curso/java/part21.html) fizemos uma chamada explícita do método **System.out.println** para mostrar um texto na tela do computador. As chamadas implícitas ocorrem quando o interpretador Java chama um método por sua própria deliberação. A chamada do método **main** é um exemplo de chamada impícita. O interpretador Java chama esse método para dar início à execução do programa.

**Declarando métodos**

A declaração mais simples que podemos fazer de um método (lembrando que isso deve ser feito dentro de uma classe) é a seguinte:

void [nome do método] () {

[corpo do método]

}

onde o **[nome do método]** é um identificador que define o nome pelo qual o método é conhecido, e **[corpo do método]** consiste de uma lista ordenada de eclaração de variáveis, de expressões e de comandos. A primeira palavra-chave, **void**, define o valor retornado pelo método, neste caso, nenhum. Podemos usar qualquer tipo de dado válido como valor de retorno de um método. Nesse caso, ao terminar, o método seria obrigado a devolver um dado do tipo especificado. Por exemplo,

class Numero {

double x = 1;

void print() {

System.out.println("O valor e " + x);

}

}

define uma classe chamada **Numero**, a qual contém uma variável **x**, inicializada com 1, e um método sem valor de retorno, **print**,que apenas escreve um texto e o valor de **x**, através da chamada do método **System.out.println**.

O par de parênteses adiante do nome do método introduz uma lista (vazia, neste caso) de argumentos. A chamada de um método pode ser acrescida de parâmetros, os quais são associados aos seus respectivos argumentos.

Um exemplo de métodos que retornam valores é o seguinte:

class Calculador {

int Soma(int a, int b) {

return a + b;

}

double Produto(double a, double b) {

return a \* b;

}

}

O primeiro método, **Soma**, realiza a adição de de dois números inteiros fornecidos pelos argumentos **a** e **b**, devolve a soma valor de retorno. O segundo método realiza a multiplicação de dois números de ponto-flutuante **a** e **b** devolvendo seu produto.

A sintaxe completa para a declaração de um método é a seguinte:

[moderadores *de acesso*] [*modificador*] [tipo do valor de retorno] [nome] ([argumentos*]*)  
*throws [lista de excessões]*{ [corpo]

}

onde os termos em itálico são opcionais (ou acessórios). Neste capítulo, vamos analisar detalhadamente cada um dos termos [[moderadores de acesso]](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#met_acesso), [[modificador]](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#met_mod), [[tipo do valor de retorno]](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#met_retorno), e [[argumentos]](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#met_param). Vamos, porém, adiar um pouco as explicações sobre [lista de excessões] até o capítulo sobre [excessões](http://www.dm.ufscar.br/~waldeck/curso/java/part28.html). Uma excessão à esta sintaxe é a que se aplica aos métodos especiais, chamados construtores, que serão vistos adiante no capítulo sobre [classes](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html#construtores).

**Moderadores de Acesso**

Os moderadores de acesso são empregados para restringir o acesso a um método. Entretanto, independentemente do moderador escolhido, um método é sempre acessível, isto é, pode ser chamado, a partir de qualquer outro método contido na mesma classe. Os moderadores de acesso existentes em Java são os seguintes:

* **public**: O método declarado com este moderador é público e pode ser chamado a partir de métodos contidos em qualquer outra classe. Esta é a condição de menor restrição possível.
* **protected**: O método é protegido pode ser chamado por todas as classes que compõe um conjunto maior chamado **package**. Veremos adiante que as packages são um modo conveniente de organizar diversas classes que estão em estreito relacionamento.   
    
  **Obs:** é importante avisar que você pode ter problemas em identificar violações com respeito à chamada de métodos protegidos. Isso se deve ao fato do compilador não sinalizar esse fato precisamente, isto é, a tentativa de chamar um método protegido a partir de uma classe que não faz parte do **package**. Ao invés disso a mensagem poderá se parecer com a seguinte:

No method matching funcao() found in class Matematica

* **friendly**: Uso do método é permitido dentro da classe que o contém, assim como dentro de qualquer classe que tenha sido derivada dessa classe, ainda que esteja fora do **package**. Este é o moderador default, isto é, aquele que é assumido se nenhum moderador for explicitamente especificado.
* **private**: O método é privativo da classe que o contém e seu uso é vedado a qualquer outra classe.
* **private protected**: o método é acessível pela classe que o contém, assim como por qualquer classe que tenha sido derivada dessa classe. Porém, isso somente é permitido apenas dentro de um mesmo arquivo-fonte.

Exemplo:

// classe de numero

class numero {

double x=1;

public void print1() {

System.out.println("O valor e "+x);

}

void print2() {

System.out.println("O valor e "+x);

}

}

// classe principal

public class PrintNum {

public static void main(String args[]) {

numero num = new numero();

num.print1(); // correto

num.print2(); // ilegal

}

}

O exemplo acima dará um erro, pois não pode acessar o **print2**. O método **print1** é definido como **public** e portanto, está disponível a qualquer classe, mas o **print2** não tem especificação (logo, é assumido como **friendly**) e portanto, a classe principal PrintNum não pode acessa-lo.

**Modificador do Método**

O modificador do método permite especificar algumas propriedades de um método, determinando como classes derivadas podem ou não redefinir ou alterar o método, e de que forma esse método será visível.

* **static:** o método é compartilhado por todos os objetos instanciados a partir da mesma classe. Um método static não pode acessar qualquer variável declarada dentro de uma classe (salvo se a variável estiver declarada também como **static**, o que veremos mais adiante), pois não é capaz de dicernir entre os diferentes objetos que compartilham esse método.
* **abstract**: Podemos declarar um método sem contudo especificar seu corpo, dizendo-o abstrato. Isto funciona como uma espécie de lembrete para que alguma classe derivada complete a declaração fornecendo um corpo. Assim sendo, uma classe que contenha um método abstrato, ou que seja derivada de alguma classe que contenha um método abstrato mas não complete sua declaração, não pode ser instanciada. O uso da palavra-chave **abstract** é opcional, isto é, as duas declarações seguintes são equivalentes:

abstract void print();

void print();

* **final**: Especifica que nenhuma classe derivada pode alterar ou redefinir este método. Um método declarado como final deve ser obrigatóriamente seguido de um corpo.
* **native**: Declara apenas o cabeçalho sem corpo, como no caso de **abstract**. Porém, esta especificação designa um método que implementado em outra linguagem como C++, por exemplo.
* **synchronized**: Esta declaração é usado para desenvolver o programa de *processamento concorrente*. Seu propósito é impedir que dois métodos executando concorrentemente acessem os dados de uma classe ao mesmo tempo. **synchromized** especifica que se um método estiver acessando o dado, outros que também desejem acessá-lo têm que esperar.

**Tipo de Valor de Retorno:**

O tipo de valor de retorno é especificado por uma palavra chave ou nome de classe na declaração do método, estabelecendo o valor que ele pode devolver.

// Classe de números complexos

class Complexo {

double x, y; // parte real e complexo, respectivamnte

public double Re() { // retorna a parte real

return x;

}

public double Im() { /retorna a parte imaginaria

return y;

}

public Complexo Vezes(Complexo c) {

Complexo resultado;

resultado.x = x \* c.x - y \* c.y;

resultado.y = x \* c.y + y \* c.x;

return resultado;

}

public void print() {

System.out.println("(" + x + " + " + y + "i)");

}

}

public class Teste {

public static void main(String args[]) {

Complexo z, w; z.x = 1;

z.y = 2;

System.out.print( "O valor de z é ");

z.print();

System.out.println( "A parte real de z é = " + z.Re() );

System.out.println( "A parte imaginária de z é = ", z.Im() );

System.out.print("O valor de z ao quadrado é ");

w = z.Vezes( z );

w.print();

}

}

Ao executar esse programa teríamos a resposta:

O valor de z é (1 + 2i)

A parte real de z é = 1

A parte imaginária de z é = 2

O valor de z ao quardado é (-3 + 4i)

Um método que retorna valor, isto é, não declarado como **void**, deve conter a linha **return ...;** a qual especifica o valor a ser retornado. Por exemplo, **return x;** especifica que o valor da variável **x** será retornado.

**Lista de Argumentos:**

A lista de argumentos é a lista de valores que o método vai precisar, obdecendo a sintaxe

[tipo 1] [nome 1], [tipo 2] [nome 2], ...

onde [tipo ?] é a especificação do tipo de dado e [nome ?] é um identificador pelo qual o parâmetro é conhecido.

Exemplo:

// Classe de numeros complexos

class Complexo {

double x=1, y=1; /\* parte real e complexo, respectivamnte

public double Re() { // retorna a parte real

return x;

}

public double Im() { /retorna a parte imaginaria

return y;

}

public set(double a, b){

x = a;

y = b;

}

}

public class Teste {

public static void main(String args[]) {

Complexo c = new Complexo();

c.set(3,2);

System.out.println("z = (" + c.Re() + " + " + c.Im() + "i)");

}

}

uma observação importante é que Java trata o objeto por referência e por isso, se o método modificar o objeto recebido como parâmetro, o objeto será modificado externamente. No entando, se o parâmetro recebido for tipo simples (int, double, char, float, etc), visto no [tipo de dados](http://www.dm.ufscar.br/~waldeck/curso/java/part22.html), o método pode alterar o parâmetro a vontade que não influencia no valor externo.

Para ilustrar, veja o exemplo a seguir:

// classe de inteiro

class inteiro {

public int i;

}

// classe principal

public class TestPar {

// método que altera o valor do parametro int

// este metodo deve ser static, pois sera chamado

// pelo metodo main() que e static

static void MudaInt(int k) {

System.out.println("MudaInt: valor de k e " + k);

k += 2;

System.out.println("MudaInt: valor de k e apos incremento e " + k);

}

// método que altera o valor do parametro inteiro

// este metodo deve ser static, pois sera chamado

// pelo metodo main() que e static

static void MudaInteiro(inteiro k) {

System.out.println("MudaInteiro: valor de k e " + k.i);

k.i += 2;

System.out.println("MudaInteiro: valor de k e apos incremento e " + k.i);

}

// main() do TestPar

public static void main(String args[]) {

int i;

inteiro n = new inteiro();

i = 2;

n.i = 3;

System.out.println("Valores originais:");

System.out.println("Valor de i e "+i);

System.out.println("Valor de n e "+n.i);

MudaInt(i);

MudaInteiro(n);

System.out.println("Valores apos a chamada dos metodos:"):

System.out.println("Valor de i e "+i);

System.out.println("Valor de n e "+n.i);

}

}

A especificação **public** de uma variável dentro da classe, faz com que este variável seja acessado de qualquer lugar. Para especificar a variável **i** do objeto **k**, basta escrever **k.i**. Em geral, a variável **[var]** dentro do objeto **[obj]** do tipo classe e referenciado por **[obj].[var]**. Note o uso de especificação **static** para os métodos **MudaInt()** e **MudaInteiro()**. Esta especificação é necessaria, pois o método principal é **static**, e método **static** não pode chamar outros métodos da mesma classe que não seja **static**.

O exemplo acima fornece a saída:

Valores originais:

Valor de i e 2

Valor de n e 3

Valores apos a chamada dos metodos:

Valor de i e 2

Valor de n e 5

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), [seguinte](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html)]

Expressões são combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenar strings, comparar valores, realizar operações lógicas e manipular objetos. Sem expressões, uma linguagem de programação seria praticamente inútil. O resultado da avaliação de uma expressão é em geral um valor compatível com os tipos dos dados que foram operados.

A maneira como uma expressão é avaliada (ou calculada) obdece as mesmas regras familiares da Matemática. A regra mais simples é a da associatividade. Quando um mesmo operador aparece mais de uma vez em uma expressão, como em **a+b+c**, então o operador mais à esquerda é avaliado primeiro e em seguida o da direita, e assim por diante. Esta seria então equivalente a **((a+b)+c)**.

**Precedência**

Há situações em que é necessário juntar operadores diferentes numa mesma expressão. Nesse caso a associatividade não se aplica mais trivialmente. Nesse caso como descobrir quais são os operandos de um operador, e quais operadores são avaliados primeiro? A resposta está na *precedência* dos operadores. Por exemplo, excrevendo **a+b\*c**, somos levados naturalmente a multiplicar primeiro **b** com **c** e em seguida o somar o produto com **a**. Isto porque a multiplicação tem precedência sobre a adição. A precedência é então um conjunto de regras que permitem determinar quais são os operandos de um dado operador.

Como a linguagem Java é rica em operadores, alguns pouco familiares, precisamos conhecer a precedência desses operadores.

Na tabela a seguir, está indicada a pecedência dos operadores comumente usados em Java. A tabela apresenta os operadores ordenados da maior precedência para a menor. Observe que, existem alguns operadores que naturalmente não requerem preocupação quanto à sua precedência, devido a forma como são empregados.

|  |  |
| --- | --- |
| **Operador** | **Descrição** |
| . [] () (*tipo*) | Máxima precedência: separador, indexação, parâmetros, conversão de tipo |
| + - ~ ! ++ -- | Operador unário: positivo, negativo, negação (inversão bit a bit), não (lógico), incremento, decremento |
| \* / % | Multiplicação, divisão e módulo (inteiros) |
| + - | Adição, subtração |
| << >> >>> | Translação (bit a bit) à esquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0) |
| < <= >= < | Operador relacional: menor, menor ou igual, maior ou igual, maior |
| == != | Igualdade: igual, diferente |
| & | Operador lógico **e** bit a bit |
| ^ | Ou exclusivo (**xor**) bit a bit |
| | | Operador lógico **ou** bit a bit |
| && | Operador lógico **e** condicional |
| || | Operador lógico **ou** condicional |
| ?: | Condicional: **if-then-else** compacto |
| = op= | Atribuição |

**Conversão entre tipos de dados**

Ao trabalhar com expressões, salvo quando todos os operando são do mesmo tipo, é inevitável ter que considerar conversões entre um tipo de dado e outro. Há basicamente dois tipos de conversões de dados. O primeiro se refere a conversão implicita na qual, os dados são convertidos automaticamente, praticamente sem a preocupação do programador. Isto ocorre no caso de conversão de dados de tipo interio para real e de números para strings. Por exemplo:

double x;

int i = 20;

x = i;

Neste caso o valor do inteiro **i** é convertido automaticamente para um double antes de ser armazenado na variável **x**. As regras de conversão implícita empregadas pela linguagem Java são as seguintes:

* os operadores unários **++** e **--** convertem um tipo **byte** e **short** são convertidos para um **int**, e os demais tipos não são afetados
* para os operadores binários, as regras são um pouco mais complicadas. Para operações envolvendo apenas inteiros, se um dos operandos for **long**, o outro será convertido para um **long** antes de realizar a operação, a qual resultará num **long**. Caso contrário, ambos os operandos são convertidos para um **int** e o resultado será também um **int**, a menos que o resultado da operação seja grande demais para caber num **int**. Nesse caso, o resultado será convertido para um **long**. Para operações envolvendo números de ponto flutuante, se um dos operadores for **double**, o outro será convertido para **double** antes de realizar a operação e o resultado será um **double**. Do contrário, ambos os operando são convertidos para **float**, e o resultado será um **float**.

Algumas vezes, porém, as conversões implícitas não são suficientes para garantir um resultado esperado em uma expressão. Nesses cados, é importante podermos controlar precisamente a ocorrência de uma conversão de tipo. Isto pode ser feito por meio de um operador unário de conversão. Por exemplo:

float eventos = 25.7;

float dias = 7.2;

x = (int)(eventos / dias);

O resultado da expressão acima será precisamente 3, isto é a parte inteira de 25.7 dividido por 7.2. A diferença é que o operador de conversão de tipo **(int)** transformou o valor do quociente 25.7/7.2 em um inteiro, truncando as casas decimais. Note que o operador de conversão é um operador unário de maior precedência, logo, tivemos de colocar a divisão entre parênteses para garantir que o resultado dessa divisão formasse um único operando. A conversão entre quaisquer inteiros e entre inteiros e números de ponto flutuante é permitida. Porém, não é permitido converter dados do tipo **boolean** para nenhum outro tipo, enquanto a conversão entre objetos somente é permitida para classes parentes.

Vejamos ainda um outro exemplo de utilização do operador de conversão de tipo:

int a = 3;

int b = 2;

double x;

x = a / b;

Neste exemplo, desejamos que a variável **x** tenha o valor do quociente entre **a** e **b**. Como **x** é um dado de ponto flutuante, presume-se que o resultado desejado neste exemplo seja, digamos, **x=3/2=1.5**. Como porém os operandos da divisão são ambos inteiros, o resultado será também um inteiro, isto é, teremos **x = 1.0**. Para contornar essa situação, podemos converter explicitamente um dos operandos para um dado em ponto flutuante, fazendo:

x = (double)a / b;

Observamos que a conversão entre inteiros de maior comprimento para inteiros de menor comprimento pode causar perda de informação na representação dos valores. Caso o valor a ser convertido ocupe mais bits do que o tipo da variável recebendo esse valor, então o valor é truncado.

int l = 393;

byte b;

b = l;

System.out.println("O valor de b é " + b);

Neste exemplo teremos como resultado:

O Valor de b é -119

Por que isso aconteceu? Fica mais fácil de compreender isso se interpretarmos os números acima na notação binária. O número 393 é representado em base dois pelo número 110001001, que tem 9 bits de comprimento (omitimos os zeros não significativos). Como a variável **b** é do tipo **byte** e seu comprimento é de apenas 8 bits, teremos os 8 primeiros bits da direita para a esquerda atribuídos a **b**, isto é, 10001001, que representa o número -119 em base dois, na notação de complemento de dois.

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), [seguinte](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html)]

* [Execução Condicional](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "cond)

* [if-then-else compacto](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "ifcompact)

* [Execução seletiva de múltiplos comandos](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "select_cond)

* [Execução seletiva por valores](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "select)

* [Laço de iteração](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "loopwhile) *[enquanto/faça](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "loopwhile)*

* [Laço de iteração](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "loopdowhile) *[faça/enquanto](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "loopdowhile)*

* [Laço de iteração com contagem](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "loopfor)

* [break e continue](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "break)

Controle de fluxo é a habilidade de ajustar a maneira como um programa realiza suas tarefas. Por meio de instruções especiais, chamadas *comandos*, essas tarefas podem ser executadas seletivamente, repetidamente ou excepcionalmente. Não fosse o controle de fluxo, um programa poderia executar apenas uma única seqüência de tarefas, perdendo completamente uma das características mais interessantes da programação: a dinâmica.   
  
Podemos classificar os comandos aceitos pela linguagem Java em basicamente quatro categorias:

|  |  |
| --- | --- |
| **Comando** | **Palavras-chave** |
| Tomada de decisões | **if**-**else**, **switch**-**case** |
| Laços ou repetições | **for**, **while**, **do**-**while** |
| Apontamento e tratamento de excessões | **try**-**catch**-**finally**, **throw** |
| outros | **break**, **continue**, ***label*:**, **return** |

**[Execução Condicional](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_cond)**

A forma mais simples de controle de fluxo é o comando **if-else***.* Ele é empregado para executar seletivamente ou condicionalmente um outro comando mediante um critério de seleção. Esse critério é dado por uma expressão, cujo valor resultante deve ser um dado do tipo booleano, isto é, **true** ou **false**. Se esse valor for **true**, então o outro comando é executado; se for **false**, a excecussão do programa segue adiante. A sintaxe para esse comando é

if ([condição])

   [comando]   // Executado se a condição for true

Uma variação do comando **if-else** permite escolher alternadamente entre dois outros comandos a executar. Nesse caso, se o valor da expressão condicional que define o critério de seleção for **true**, então o primeiro dos outros dois comandos é executado, do contrário, o segundo.

if([condição])

  [comando 1]   // Executado se a condição for true

else

  [comando 2]   // Executado se a condição for false

Por exemplo:

import java.io.StreamTokenizer;

public class sqrt {

  public static void main(String args[]) {

    double x;

StreamTokenizer inp = new StreamTokenizer(System.in);

    // Lê um dado double a partir do teclado

    System.out.print("x = "); System.out.flush();

try {

inp.nextToken();

x = inp.nval;

}

catch(java.io.IOException e) {

x = 0;

}

    // Decide se é possível calcular a raiz quadrada do

    // número dado. Sendo possível, calcula-a. Do contrário

    // emite uma mensagem de aviso

    if(x >= 0)

      System.out.println("raiz quadrada de x e " + sqrt(x));

    else

      System.out.println("x e negativo");

  }

}

E se toda uma lista de comandos deve ser executada seletivamente? A esta altura, convém dizer que há uma maneira bastante conveniente de agrupar longas seqüências de comandos formando uma unidade lógica. Trata-se da noção de *bloco de comandos*. Este consiste basicamente de uma lista de comandos delimitados por chaves { }. Para efeitos de programação, um bloco de comandos é interpretado como se fosse um único comando. Eis alguns exemplos de blocos de comandos:

1. {

System.out.print("x = ");    
System.out.println(x);    
}

1. { temp = y; y = x; x = temp; }

Desse modo, podemos fazer também a execução seletiva de blocos de comandos conforme ilustra o seguinte programa para discutir sobre as raízes de uma equação do segundo grau *ax2+bx+c=0*:

import java.io.StreamTokenizer;

public class baskhara {

public static void main(String args[]) {

double a, b, c, delta;

StreamTokenizer in = new StreamTokenizer(System.in);

// Requisitando do usuário os dados sobre a equação a ser

// analisada.

//

try {

System.out.println("Digite os coeficientes da equação ax^2+bx+c = 0");

System.out.print("a = "); System.out.flush(); in.nextToken();

a = in.nval;

System.out.print("b = "); System.out.flush(); in.nextToken();

b = in.nval;

System.out.print("c = "); System.out.flush(); in.nextToken();

c = in.nval;

}

catch(java.io.IOException e) {

System.out.println("Falha na entrada dos dados.");

a = 0;

b = 0;

c = 0;

}

// Calculando o discriminante da equação

//

delta = b\*b - 4\*a\*c;

System.out.println( "Equação: ("+a+")x^2+("+b+")x+("+c+")" );

// Decidindo sobre o número de raízes da equação mediante o

// valor do discriminante calculado acima

//

if ( delta > 0 ) { // A eq. tem duas raízes reais

double r, d, x1, x2;

r = Math.sqrt(delta);

d = 2\*a;

x1 = ( -b - r ) / d;

x2 = ( -b + r ) / d;

System.out.println("A equação tem duas soluções reais: ");

System.out.println(" x1 = " + x1);

System.out.println(" x2 = " + x2);

}

else

if ( delta < 0 ) // A eq. não tem raízes reais

System.out.println("A equação não tem raízes reais.");

else { // A eq. tem uma raiz real dupla

double x;

x = -b / (2\*a);

System.out.println("A equação tem uma única raiz real:");

System.out.println(" x1 = x2 = " + x);

}

System.out.println("Fim da discussão.");

}

}

[if-then-else compacto](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_ifcompact)

O **if-else** compacto não é propriamente um comando, mas um operador que realiza avaliação seletiva de seus operandos, mediante o valor de uma expressão booleana semelhante à do comando **if-else**. Se essa expressão for **true**, então um primeiro operando é avaliado; se for **false** então o segundo operando é avaliado. A sua sintaxe é   
  
[expressão condicional] ? [expressão 1] : [expressão 2]    
  
onde **[condição condicional]** deve resultar em **true** ou **false**, **[expressão 1]** e **[expressão 2]** são os operandos, que podem ser expressões quaisquer. O valor resultante da aplicação do operador **if-else** compacto é obviamente igual ao valor do operando que tiver sido avaliado. O tipo desse valor obedece às regras de [conversão entre tipos de dados](http://www.dm.ufscar.br/~waldeck/curso/java/part25.html#conversao) discutida anteriormente.   
  
Para melhor ilustrar o **if-else** compacto, consideremos o seguinte comando:

y = x < 1 ? x\*x : 2-x;

Este é logicamente equivalente à seguinte seqüência de comandos:

if (x<1) then

   y = x\*x;

else

   y = 2-x;

É importante notar que o **if-else** compacto é um operador de baixa precedência, logo o uso de parênteses para separar seus operandos não é necessário (a menos que mais de um desses operadores esteja presente na mesma expressão). Porém há casos em que o uso de parênteses para definir claramente os operandos é essencial. Por exemplo, **y = |x|\*sin(x)**, podeser codificado como   
  
y = ( x<0 ? -x : x ) \* Math.sin(x); // aqui os parenteses são essenciais.    
  
Sem os parênteses, **x \* Math.sin(x)** seria visto pelo operador **if-else** compacto como se fosse um único operando, devido à alta precedência do operador multiplicação.

**[Execução Seletiva de Múltiplos Comandos](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_select_cond)**

Freqüentemente, desejamos que um único comando (ou único bloco de comandos) de uma lista seja executado mediante um dado critério. Isso pode ser feito através do aninhamento ou acoplamento de vários comandos **if-else**, do seguinte modo:

if ([condição 1])

   [comando 1]

else if ([condição 2])

   [comandos 2])

else if ([condição 3])

   [comando 3]

....

else

   [comando n]

A presença do último **else**, juntamente com seu comando, é opcional. Neste código, o **[comando 1]** será executado (e os demais saltados) caso a primeira condição seja **true**, o **[comando 2]** será executado (e os demais saltados) caso a primeira condição seja **false** e a segunda condição seja **true**, e assim sucessivamente. O **[comando n]** (se houver) somente será executado (e os demais saltados) caso todas as condições sejam **false**. Vejamos um exemplo:

import java.io.StreamTokenizer;

public class sqrt {

  public static void main(String args[]) {

    double x, y;

StreamTokenizer inp = new StreamTokenizer(System.in);

    /\* Desejamos definir y tal que

           | x+2,  se x < -1,

       y = | 1,    se -1 <= x < 1,

           | x\*x,  se 1 <= x.

    \*/

try {

    System.out.print("x = "); System.out.flush(); inp.nextToken();

    x = inp.nval;

}

catch(java.io.IOException e){

x = 0;

}

    if (x<-1)

       y = x+2;

    else

       if(x<=0)

          y = 1;

       else

          y = x\*x;

    System.out.println("y = " + y);

  }

}

[Execução Seletiva por Valores](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_select)

Assim como no caso execução seletiva de múltiplos comandos, há situações em que se sabe de antemão que as condições assumem o valor true de forma mutuamente exclusiva, isto é, apenas uma entre as condições sendo testadas assume o valor **true** num mesmo momento. Nesses casos, a linguagem Java (como também as linguagem C, C++ e Pascal) provê um comando de controle de fluxo bastante poderoso. Trata-se do comando **swich**, cuja sintaxe é a seguinte:

switch([expressão]) {

  case [constante 1]:

     [comando 1]

     break;

  case [constante 2]:

     [comando 2]

     break;

     .

     .

     .

  case [constante n]:

     [de comando n]

     break;

  default:

     [comando]

}

A **[expressão]** pode ser qualquer expressão válida. Esta é avaliada e o seu valor resultante é comparado com as constantes distintas **[constante 1]**, **[constante 2]**, ..., **[constante n]**. Caso esse valor seja igual a uma dessas constantes, o respectivo comando é executado (e todos os demais são saltados). Se o valor for diferente de todas essas constantes, então o comando presente sob o rótulo **default:** é executado (e todos os demais são saltados), caso este esteja presente. Por exemplo:

public class test {

  public static void main(String args[]) {

    int op;

    op = 2;

    System.out.print("valor de op eh: "+op)

    switch(op) {

      case 1:

         System.out.println("case 1: op="+op);

         break;

      case 2:

         System.out.println("case 2: op="+op);

         break;

      case 3:

         System.out.println("case 3"+op);

         break;

      default:

         System.out.println("default: op nao esta no limite 1..3");

    }

  }

}

**[Laço de iteração](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_loopwhile) *[enquanto/faça](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_loopwhile)***

Freqüentemente, desejamos que uma tarefa seja executada repetidamente por um programa enquanto uma dada condição seja verdadeira. Isso é possível pela utilização do comando **while**. Este comando avalia uma expressão condicional, que deve resultar no valor **true** ou **false**. Se o valor for **true**, então o comando subjacente é executado; se a expressão for **false**, então o comando é saltado e a execução prossegue adiante. A diferença é que após executar o comando subjacente, a expressão condicional é novamente avaliada e seu resultado novamente considerado. Desse modo a execução do comando subjacente se repetirá, até que o valor da expressão condicional seja **false**. Observe, porém, que a expressão é avaliada antes de uma possível execução do comando subjacente, o que significa que esse comando pode jamais ser executado.

while([condição])

   [comando subjacente]

Por exemplo:

// Achar o raiz quadrada de 2 pelo metodo de bissecção

public class sqrt2 {

  public static void main(String args[]) {

    double a, b, x=1.5, erro = 0.05;

    a = 1; b = 2; // 1 < (raiz de 2) < 2

    while( (b-a) > erro ) {

       x = (a+b)/2;

       if (x\*x < 2) // x < raiz de 2

          a = x;

       else // x >= raiz de 2

          b = x;

    }

    System.out.println("Valor aproximado de raiz quadrada de 2: " + x);

 }

}

Uma das observações importantes é sempre certificar que não ocorra o laço infinito (um laço que nunca para, pois a condição sempre é verdadeira). Caso tenha alguma chance de entrar no laço infinito, coloque um contador ou user o laço for.

**[Laço de iteração](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_loopdowhile) *[faça/enquanto](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_loopdowhile)***

Um outro tipo de laço de repetição, similar ao enquanto/faça, é o laço *faça/enquanto.* Este é introduzido por um par de comandos **do**/**while** que tem a seguinte sintaxe:

do

   [comando]

while([condição]);

Diferente do laço enquanto/faça, este tipo de laço de repetição executa o comando e em seguida avalia a expressão condicional. A repetição ocorre se o valor dessa expressão for **true**. Se esse valor for **false**, a execução prossegue adiante do **while**. Vejamos o seguinte exemplo:

public class Menu {

public static void main(String args[]) {

char op;

int i = 0;

double x = 0;

do {

System.out.println("\nOpcoes:");

System.out.println("p - Atribuir: x = 0.5, i = 2");

System.out.println("n - atribuir: x = -0.2, i = -1");

System.out.println("x - ver o valor de x");

System.out.println("i - ver o valor de i");

System.out.println("f - fim");

System.out.print("Sua escolha: "); System.out.flush();

try {

op = (char)System.in.read();

System.in.read(); // Para pegar o 'enter'

}

catch(java.io.IOException e) {

op = 'q';

}

switch(op) {

case 'p':

x = 0.5; i = 2;

break;

case 'n':

x = -0.2; i = -1;

break;

case 'x':

System.out.println("\n--> x = " + x);

break;

case 'i':

System.out.println("\n--> i = " + i);

break;

case 'f':

break;

default:

System.out.println("\n--> Opcao invalida");

}

} while(op != 'f');

System.out.println("\nAte logo.");

}

}

**[Laço de iteração com contagem](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_loopfor)**

Em certas situações, precisamos de laços de repetições nos quais alguma variável é usada para contar o número de iterações. Para essa finalidade, temos o laço **for**. Este é o tipo de laço mais geral e mais complicado disponível na linguagem Java. Sua sintaxe é a seguinte:

for ([expressão 1]; [condição]; [expressão 2])

 [comando]

Onde [expressão 1] é chamada *expressão de inicialização*, [condição] é uma expressão condicional e [expressão 2] é uma expressão qualquer a ser executado no final de cada iteração. O laço for acima é equivalente a:

[expressão 1]

while ([condição]) {

    [comando]

    [expressão 2]

}

Isto que dizer que o laço **for** avalia inicialmente a expressão de inicialização. Em seguida, avalia a expressão condicional. Se o valor desta for **true**, então o comando é executado,  a segunda expressão é avaliada em seguida, e finalmente o laço volta a avaliar novamente a expressão condicional. Do contrário, se o valor da expressão for false, a execução prossegue adiante do laço **for**. Este arranjo é muito conveniente para manter variáveis de contagem, conforme ilustra o seguinte exemplo:

for (i=0; i<n; i++)

   System.out.println("V["+i+"]="+v[i]);

Este código imprime os valores de cada elemento **v[i]** de um vetor **v**, para i variando de **0** até **n-1**. O operador **++** foi utilizado na ultima expressão do laço **for** para somar um ao valor da variável de contagem. Caso necessitássemos de um incremento (ou decremento) maior do que um, poderíamos usar os operadores **+=** ou **-=**. Por exemplo, para imprimir todos os números pares entre de **0** até **10**, poderíamos fazer:

for(i=0; i<=10; i+=2)

   System.out.println(" " + i);

Tanto a **[expressão 1]** quanto a **[expressão 2]** do laço **for** permitem acomodar múltiplas expressões, bastando separá-las por vírgula. Por exemplo, a soma de {1/n} n=1,2, .., N pode ser obtida por:

for (soma=0, n=1; n<=N;n++)

   soma += 1/n;

ou ainda por   
  
for(soma=0, n=1; n<=N; soma += 1/n, n++);

**[break e continue](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html" \l "i_break)**

O comando **break** é usado para interromper a execução de um dos laços de iteração vistos acima ou de um comando **switch**. Este comando é comumente utilizado para produzir a parada de um laço mediante a ocorrencia de alguma condição específica, antes da chegada do final natural do laço.   
  
Exemplo:

// Achar i tal que v[i] é negativo

for(i=0; i<n;i++)

  if(v[i] <0) break;

if(i == n)

  system.out.println("elemento negativo não encontrado.");

E se isto se der dentro de um laço duplo? Nesse caso, o comando **break** provocará a interrupção apenas do laço em que o comando é imediatamente subjacente. Os outros laços continuam normalmente.    
  
Exemplo:

// Notificando a existencia de elemento nulo em cada linha do matriz A

for(i=0; i<m; i++)        // Para cada linha i da matriz faça

  for(j=0; j<n; j++)      // Para cada coluna j da matriz faça

    if(A[i][j] == 0) {

       System.out.println("A possui elemento nulo na linha "+i);

       break;

     }

O comando **continue** tem a função de pular direto para final do laço, mas em vez de interromper o laço como no **break**, ele continua executando o próximo passo do laço. Não vamos ficar estudando o uso de continue por ser puco usual na programação estruturada.   
  
O comando "goto" em Java é feito pelo **break** ou **continue** com rótulo. Isto é útil para interromper de uma vez só, a execução de um laço multiplo ou pular para um determinado ponto do programa. O uso de saltos deste tipo deve ser evitado na programação estruturada, exceto quando trata de alguma otimização muito importante. Caso contrário, nosso programas pode tornar totalmente ilegível. Nós vamos ver um exemplo, mas lembre-se que se necessitar de um "goto", está faltando a estruturação do programa. Nós estudamos essas coisas para poder entender o programa já feito.   
  
exemplo:

// calculando a raiz da soma dos quadrados de A[i][j]

soma = 0;

for(i=0;i<m;i++)

 for(j=0;j<n;j++)

   if(A[i][j]<0) {

      System.out.println("Erro. existe elemento negativo.");

      break fim;

   }

   esle

      soma += Math.sqrt(A[i][j]);

fim:

System.out.println("soma = " + soma);

o nome colocado na localização é denominado de rórotulo e deve ser declarado como **label**.

[**Exercícios**](http://www.dm.ufscar.br/~waldeck/curso/java/part26.html#i_break)

1. Implemente os exemplos desta seção. Eventualmente alguns exemplos necessitarão ser adequadamente completados para poderem executar.

### Introdução

No capítulo sobre [métodos](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#classes), antevimos alguns conceitos básicos sobre classes. Agora, vamos aprofundar mais esses conceitos, permitindo-nos elaborar classes mais sofisticadas, com toda a funcionalidade que elas permitem.

Usamos as classes para construir objetos, o que é chamado de *instanciação*. E os objetos consistem a essência da programação orientada a objetos (ou *OOP*, do inglês *Object-Oriented Programming*). Falando intuitivamente, as classes consistem de uma maneira de organizar um conjunto de dados, e designar todos os métodos necessários para usar ou alterar esses dados.

O conjunto de todos os dados contidos em uma classe definem o *estado* de um objeto. Por exemplo, se tivéssemos uma classe Semaforo contendo uma única variável chamada VermelhoVerdeAmarelo, então o estado de Semaforo é determinado pelo valor da de VermelhoVerdeAmarelo.

public class Semaforo {

   int VermelhoVerdeAmarelo = 0; // 0=vermelho,1=verde,2=amarelo

   void Alternar() {

      VermelhoVerdeAmarelo = ++VermelhoVerdeAmarelo % 3;

   }

}

Os métodos de uma classse, por sua vez, determinam a utilidade que uma classe terá. No caso da classe Semaforo, seu único método Alternar tem como função provocar a mudança da luz de vermelho a verde, de verde a amarelo e de amarelo a vermelho, respectivamente, em cada nova chamada. Assim, se o método Alternar for chamado em intervalos de tempo regulares, poderemos utilizar o estado da classe Semaforo para controlar um semáforo com luzes reais.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngPara distinguir entre variáveis declaradas em classes daquelas declaradas localmente dentro de métodos, comumente nos referimos àquelas como *campos*. Assim, dizemos que VermelhoVerdeAmarelo é um campo da classe Semaforo.

#### Herança

Contudo, uma das maiores vantagens da OOP reside na possiblidade de haver *herança* entre classes. Esta consiste na capacidade de construir novas classes a partir de outras existentes. Nesse processo, os dados e métodos de uma classe existente, chamada *parente* (ou *superclass*), são herdados pela nova classe, chamada *subclasse*, ou *classe derivada*.

#### Encapsulamento

Outro benefício importante da OOP reside no chamado *encapsulamento*. Este consiste na habilidade de efetivamente isolar informações do restante do programa. Isto traz uma série de vantagens. Uma vez concluída uma classe intricada, por exemplo, é virtualmente possível esquecermos suas complicações internas, passando a tratá-la através de seus métodos. Ainda que mais tarde seja preciso realizar mudanças significativas no interior de uma classe, não será necessário modificar o restante do programa. Outro benefício desta prática é garantir que a informação não será corrompida acidentalmente pelo resto do programa. Criamos, assim, programas mais robustos e confiáveis.

#### Polimorfismo

Finalmente, uma característica importante das classes reside no fato de que as subclasses de uma dada classe são consideradas do mesmo tipo de seu parente. Isto é chamado *polimorfismo*. Este permite a realização de uma mesma operação sobre diferentes tipos de classes, desde que mantenham algo em comum. Por exemplo, considere a classe Polígono e suas derivadas Retângulo e Triângulo declaradas abaixo. Apesar de retângulos e triângulos serem diferentes, eles ainda são considerados polígonos. Assim, qualquer coisa que fosse permitido fazer com uma instância (i.é, um objeto) da classe Polígono, seria também permitida para a instâncias das classes Retângulo e Triângulo. O seguinte exemplo ilustra o polimorfismo entre essas classes permitindo que se desenhe um polígono, independentemente do prévio conhecimento de que se trata de um retângulo ou de um triângulo.

class Vértice {

   public double x, y;

   Vértice( double x, double y ) {

      this.x = x;

      this.y = y;

   }

}

class Polígono {

   int numVértices;   // Quantidade de vértices do polígono

   Vértice v[];       // Coordenadas de seus vértices

}

class Gráfico {

   // Desenha o polígono no dispositivo gráfico

   void Desenhar(Polígono p) {

      SaídaGráfica g; // saída gráfica com método DesenhaLinha.

      Vértice anterior;

      anterior = p.v[0];

      // Une os vértices do polígono desenhando uma linha entre eles

      for(int i=1; i<numVértices; i++) {

         g.DesenhaLinha( anterior, p.v[i] );

         anterior = p.v[i];

      }

   }

}

class Retângulo extends Polígono {

   Retângulo(Vértice v[]) {

      this.v = new Vértice[4];

      numVértices = 4;

      for(int i=0; i<4; i++)

         this.v[i] = v[i];

   }

}

class Triângulo extends Polígono {

   Triângulo(Vértice v[]) {

      this.v = new Vértice[3];

      numVértices=3;

      for(int i=0; i<3; i++)

         this.v[i] = v[i];

   }

}

public class polimorfismo {

   // Coordenadas dos vértices de um retângulo

   static Vértice v[] = {

      new Vértice(0.0,0.0), new Vértice(2.0,0.0),

      new Vértice(2.0,1.0), new Vértice(0.0,1.0)

   };

   // Coordenadas dos vértices de um triângulo

   static Vértice w[] = {

      new Vértice(-1.0,0.0), new Vértice(1.0,0.0),

      new Vértice(0.0,1.0)

   };

   public static void main(String args[]) {

      Polígono r, t;

      Gráfico g = new Gráfico();

      r = new Retângulo(v);    // Isto é válido, pois Retângulo é um Polígono

      t = new Triângulo(w);    // Isto é válido, pois Triângulo é um Polígono

      // Desenha o retângulo

      g.Desenhar( r );

      // Desenha o triângulo

      g.Desenhar( t );

   }

}

Algumas partes deste código são novidade, como por exemplo os métodos declarados nas classes Vértice, Retângulo e Polígono, que parecem não obedecer às regras estabelecidas no capítulo sobre métodos.

### Declarando uma classe

A forma geral da declaração de uma classe é a seguinte

[*modificadores*] class [nome classe] *extends* [*nome super*]

*implements* [*nome interface*]

onde as partes que aparecem em itálico são opcionais. Como podemos notar, há quatro diferentes propriedades de uma classe definidas por essa declaração:

* Modificadores
* Nome de classe
* Super classes
* Interfaces

Vamos ver em detalhes cada uma destas propriedades a seguir.

### Modificadores

Os modificadores de uma classe determinam como uma classe será manipulada mais tarde no decorrer do desenvolvimento do programa. Estes são muito parecidos com os moderadores de acesso introduzidos anteriormente no capítulo sobre [métodos](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html#met_acesso).

Ao declarar uma nova classe, é possivel especificar um dos seguintes modificadores: public, final, abstract.

* public: permite definir classes públicas. Estas classes são acessíveis a partir de qualquer objeto, independentemente do package. Uma classe pública deve ser a única classe desse tipo no arquivo em que está declarada e o nome do arquivo deve ser igual ao da classe.
* friendly: se nenhum modificador de classe for especificado, então a classe será considerada *friendly*. Apenas os objetos integrantes do mesmo package podem utilizar uma classe friendly.
* final: Uma classe final pode ser instanciada, mas não pode ser derivada, isto é, não pode ser superclasse de nenhuma subclasse. Algumas classes predefinidas no ambiente Java têm esta propriedade. Outras não, como as classes no java.awt, por exemplo.
* abstract: Classes abstratas são aquelas que contém ao menos um método incompleto. Desse modo uma classe abstrata não pode ser instanciada, mas pode ser derivada. Neste caso, a subclasse deve prover o corpo do método para que possa ser instanciada. Isto é muito útil quando desejamos definir em uma classe regras gerais para o comportamento de uma parte do programa, para que, mais tarde, as regras mais específicas sejam introduzidas por subclasses.

### Nome de Classe

Como qualquer identificador em Java, o nome de uma classe deve obedecer às seguintes regras:

* Iniciar com uma letra, ou um dos caracteres: '$', '\_'.
* Conter somente caracteres Unicode considerados letras, dígitos ou um dos dois caracteres acima.
* Não pode ser igual a uma palavra-chave reservada pela linguagem Java, tal como void, int, for, while, etc.

Lembre-se: as letras maiúsculas e as minúsculas são consideradas diferentes.

### Super Classes

Um dos aspectos mais importantes da OOP é a capacidade de usar campos e métodos de uma classe previamente construída. Por meio da extensão de classes simples podemos construir classes maiores, acrescentando àquelas mais campos e métodos, obtendo com isto mais funcionalidades. Neste processo, há uma grande economia no esforço de codificação. Sem esse recurso, freqüentemente seria necessário recodificar grande parte dos programas para acrescentar-lhes funcionalidade ou fazer modificações significativas.

Ao derivar uma classe, estamos primeiramente fazendo uma cópia da classe parente. É exatamente isto que obtemos se deixarmos vazio o corpo da subclasse. Tal classe se comportaria exatamente como sua superclasse. Entretanto, podemos acrescentar novos campos e métodos à subclasse, além de sobrepor métodos existentes na superclasse, declarando-os exatamente como na superclasse, exceto por dar um corpo diferente.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngNão existe herança múltipla em Java. E contraste com a linguagem C++, em Java somente é possível derivar uma classe a partir de uma outra, e não de várias.

### Construtores

Os contrutores são métodos muito especiais, a começar pela sua sintaxe declarativa, e também por suas propriedades e finalidade únicas. Por exemplo, o construtor da classe Vértice vista acima é o seguinte:

Vértice( double x, double y ) {

   this.x = x;

   this.y = y;

}

Sua única finalidade é inicializar o objeto com um par de coordenadas fornecidas no momento da instanciação. Aliás, esta é a principal finalidade dos construtores: atribuir a um objeto um estado inicial, apropriado ao processamento subseqüente.

Os contrutores são métodos facilmente identificáveis pois têm o mesmo nome da classe. Além disso, os construtores não especificam nenhum valor de retorno, mesmo que este seja void, uma vez que não são chamados como os outros métodos. Os construtores somente podem ser chamados no momento da instanciação. Por exemplo:

Vértice v = new Vértice(1.0, 2.0);

Temos neste trecho de código a instanciação da classe Vértice, que ocorre no momento em que reservamos espaço para conter um novo objeto dessa classe. Nesse momento o construtor Vértice é chamado com os argumentos 1.0 e 2.0.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngÉ usual declarar os contrutores como públicos. Isto porque, se eles tiverem um nível de acesso inferior ao da classe propriamente dita, outra classe será capaz de declarar uma instância dessa classe, mas não será capaz de realizar ela mesma a instanciação, isto é, não poderá usar o operador new para essa classe. Há situações, porém, em que essa característica é desejável. Deixando seus construtores como privativos, permite a outras classes usar métodos estáticos, sem permitir que elas criem instâncias dessa classe.

Uma classe pode múltiplos construtores declarados, desde que cada um tenha lista de argumentos distinta dos demais. Isto é muito útil, pois em determinados contextos do programa um objeto deve ser inicializado de uma maneira particular em relação a outros contextos possíveis.

Quando nenhum construtor é declarado explicitamente, um construtor vazio é provido implicitamente. Por exemplo, se não tivéssemos especificado um construtor na classe Vértice, este sería o *construtor default*:

Vértice() {

}

Os construtores não podem ser declarados com os modificadores: native, abstract, static, synchronized ou final.

### Sobreposição

Não é permitido declarar em uma mesma classe dois métodos com o mesmo nome e mesma lista de argumentos. De fato, isto parece não fazer nenhum sentido, pois os métodos são unicamente identificados pelo nome e pela lista de argumentos que os acompanha. Se isso fosse permitido haveria uma grande confusão, pois como é que se poderia determinar precisamente qual método chamar?

Entretanto, uma das finalidades de permitir a derivação de classes é atribuir a elas novas funcionalidades. Isto é possível acrescentando-se novos métodos às subclasses. Mas também é possível subrepor qualquer dos métodos existentes na superclasse, declarando o novo método na subclasse exatamente com o mesmo nome e lista de argumentos, como consta na superclasse. Por exemplo, considere a classe Computador abaixo:

class Computador {

   private boolean ligado = true;

   public void Desligar() {

      ligado = false;

   }

}

Esta classe permite que o computador seja desligado, através da chamada do método Desligar. Porém, isto pode não ser muito seguro, pois poderíamos desligar o computador mesmo quando ele estiver executando algum programa. Nesse caso, podemos evitar uma catástrofe derivando a classe computador do seguinte modo:

class ComputadorSeguro extends Computador {

   private boolean executando = true;

   public void Desligar() {

      if ( executando )

         System.out.println("Há programas rodando. Não desligue!");

      else

         ligado = false;

   }

}

Agora, um objeto da classe ComputadorSeguro somente será desligado quando não tiver programas rodando (exceto quando alguém acidentalmente chutar o fio da tomada!).

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngA sobreposição somente acontece quando o novo método é declarado com exatamente o mesmo nome e lista de argumentos que o método existente na superclasse. Além disso, a sobreposição não permite que o novo método tenha mais proteções do que o método original. No exemplo acima, como o método Desligar foi declarado como public na superclasse, este não pode ser declarado private na subclasse.

### Instanciando uma classe

Uma classe define um tipo de dado. Ela não pode ser usada a não ser que seja instanciada. A exemplo dos tipos de dados primitivos os quais somente podem ser usados quando uma variável de um determinado tipo é declarada, devemos criar uma instância. Uma *instância* é um objeto do tipo definido pela classe. Qualquer classe (desde que não seja abstract) pode ser instanciada como qualquer outro tipo de dado da linguagem Java. O trecho de código abaixo exibe uma classe chamada Geometria criando um a instância da classe Vértice:

public class Geometria {

   Vértice v = new Vértice(1.2, 3.5);

   ...

}

A diferença mais evidente entre a declaração de um objeto de uma classe e a declaração de um dado primitivo reside na necessidade de reservar memória para o objeto através do uso do operador new. Na verdade, esse operador realiza uma série de tarefas:

* Reserva espaço para a instância da classe Vértice, o qual deve ser suficiente para conter seu estado, isto é, os valores dos seus campos.
* Realiza a chamada do método construtor.
* Retorna uma referência para o novo objeto (o qual é atribuído à variável v).

Outra importante diferença entre objetos e dados de tipo primitvo é que estes são sempre referenciados por valor, enquanto aqueles são sempre referenciados por meio de sua referência. Isto tem impacto significativo na maneira como os objetos são passados como parâmetros na chamada de métodos. Se o método realizar internamente alguma modificação no objeto que foi passado, essa modificação refletirá no objeto original. Isto não ocorre com a passagem de dados de tipo primitivo.

### Referindo-se às partes de uma classe

Após instanciar uma classe é desejável podermos acessar algum de seus campos ou então algum de seus métodos. Dentro de uma classe os campos e métodos são acessíveis imediatamente pelo nome. Repare como na classe [**Computador**](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html#class_comp) acima o método Desligar acessa diretamente o campo ligado, simplesmente por meio do seu nome.

Entretanto, considere a seguinte classe chamada CPD a qual contém várias instâncias da classe Computador:

public class CPD {

   Computador Gauss = new Computador(),

              Davinci = new Computador(),

              Fermat = new Computador();

...

public void Fechar() {

   Gauss.Desligar();

   Davinci.Desligar();

   Fermat.Desligar();

}

...

O método Fechar realiza o desligamento de cada particular instância da classe Computador chamando seu método Desligar. Para indicar a qual objeto o método se refere, devemos precedê-lo do nome do objeto seguido de um operador ponto '.'. A notação geral é

[nome da instância].[nome do método ou variável]

Uma excessão a essa regra aplica-se à referência de campos ou métodos declarados como static. Tais declarações são compartilhadas por todas as instâncias de uma classe, desse modo não fornecemos o nome de uma particular instância, mas o nome da própria classe ao referenciá-los.

### A especificação **this**

Vimos acima como fazer referências a partes de classes. Mas, e se desejássemos fazer referência a partes da própria classe? Isso parece evidente, porém, às vezes, o nome de um argumento ou variável declarada por um método pode coincidir com o nome de um campo da classe. Veja o exemplo da classe [**Vértice**](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html#class_vertice). Nessa classe o método construtor declara dois argumentos x e y, os quais têm o mesmo nome dos campos x e y da classe. Esse método distingue os argumentos dos campos pelo uso da especificação this. Assim this.x e this.y referem-se aos campos x e y declarados na classe, enquando x e y propriamente ditos referem-se aos argumentos do construtor. A palavra this substitui uma referência à propria classe.

Há basicamente duas situações em que devemos empregar a palavra this:

* Quando houver duas variáveis com mesmo nome numa mesma classe - uma pertencendo à classe e outra pertencendo a algum dos métodos da classe. Nesse caso, apenas esse método específico requer o uso do this se quiser fazer referência ao campo da classe.
* Quando uma classe precisa passar uma referência de si propria a um método. Vamos ter a oportunidade de explorar este aspecto quando estivermos estudando os applets.

### A especificação **super**

A palavra super provê acesso a partes de uma superclasse a partir de uma subclasse. Isto é muito útil quando estamos sobrepondo um método. Poderíamos reescrever o método Desligar da classe ComputadorSeguro do seguinte modo:

class ComputadorSeguro extends Computador {

   private boolean executando = true;

   public void Desligar() {

      if ( executando )

         System.out.println("Há programas rodando. Não desligue!");

      else

         super.Desligar();

   }

}

Note a chamada super.Desligar(). Esta corresponde a chamada do método Desligar declarado na superclasse Compudador, o qual vai efetivamente ajustar o campo ligado para o valor false. Imaginando que o método Desligar fosse muito mais complicado, não precisaríamos recodificá-lo completamente na subclasse para acrescentar a funcionalidade que permite o desligamento apenas quando o computador estiver desocupado. Basta chamá-lo da maneira prescrita.

E se o método que desejamos chamar é um construtor? Bem, nesse caso a chamada usando a palavra super bem particular. Examinemos o seguinte exemplo de uma classe chamada VérticeNumerado que estende a classe Vértice, acrescentando às coordenadas do vértice um rótulo numérico que o identifica visualmente:

class VérticeNumerado extends Vértice {

   int numero;

   VérticeNumerado( int numero, int x, int y ) {

      this.numero = numero;

      super(x, y);

   }

}

Note que a chamada super(x, y) se traduz na chamada do construtor Vértice(x,y) da superclasse. Com isto, evitamos de ter que recodificar no novo construtor as tarefas contidas no construtor da superclasse: basta chamá-lo. Vale observar que esse tipo de chamada também só é permitida de dentro de um construtor.

### Campos e Variáveis Locais

Chegou o momento de discutirmos sobre a forma como as variáveis são declaradas dentro de um programa. Podemos declarar variáveis de classe, chamadas *campos*, e variáveis de métodos, ditas *locais*.

A esta altura, já devemos ter feito a seguinte indagação: se os campos são acessíveis por qualquer dos métodos declarados em uma classse e eventualmente por métodos de classes derivadas, por que não declararmos todos os dados empregados por uma classe como campos? Uma resposta imediata a essa pergunta seria: isso provocaria um significativo desperdício de memória, pois os campos existem durante todo período de existência de um objeto. Entretanto, os dados declarados localmente por um método existem somente enquanto esse método estiver sendo executado, de modo que o espaço de memória previamente ocupado por eles é reaproveitado quando o método termina sua execução.

A capacidade de acessar uma variável de uma classe depende fundamentalmente de duas coisas: moderadores de acesso e localização da variável dentro da classe. As variáveis locais somente são acessíveis pelo método que as declara, enquanto que os campos dependem dos moderadores. Apesar de ser possível deixar todos os campos de uma classe publicamente acessíveis, isto não é recomendável. Do contrário estaríamos desperdiçando o sofisticado mecanismo de proteção de dados fornecido pela OOP, o qual permite escrever programas mais robustos e livres de erros (vulgarmente chamados *bugs*).

Os possíveis moderadores empregados na declaração de campos são os seguintes:

* friendly: todos os campos são friendly por default. Isto significa que são acessíveis por outras classes integrantes do mesmo package e não são acessíveis por nenhuma classe ou subclasse exterior ao package.
* public: idêntico ao moderador de acesso dos métodos. O campo é acessível a partir de qualquer outra classe, independentemente do package.
* protected: os campos protected podem ser acessados a partir de qualquer classe derivada da classe atual, mas não são acessíveis de fora do package.
* private: é o maior grau de proteção. Um campo private é acessível unicamente pela classe atual.
* private protected: a combinação dos moderadores private e protected estabelece que o campo seja acessível pela classe atual e por suas subclasses.
* static: Um campo static é compartilhado por todas as instâncias de uma classe, isto é, há um único valor para esse campo, independentemente da quantidade de instâncias existentes, mesmo que não haja nenhuma.
* final: um modificador final precedendo um campo declara esse campo como uma constante. Seu valor não pode mudar durante a execução do programa. Por isso, é necessário que haja uma inicialização de campo. Por exemplo:

final int MaxDimen = 10;

O uso de constantes dentro de um programa torna-o mais facilmente legível e fácil de seguir. Para economizar memória, é recomendável também declarar constantes como static.

### Classes Especiais

A linguagem Java provê algumas classes básicas para formar uma base sólida para todas as demais classes, inclusive aquelas criadas pelo programador. Dentre essas classes, seguramente as mais importantes são as classes Object, Class e String.

#### A classe **Object**

A classe Object é uma classe que serve de superclasse para todas as classes existentes em Java. Isso significa que ao criar uma classe, se não for especificada nenhuma superclasse após a palavra extends, então a classe Object será assumida automaticamente como superclasse. Portanto toda classe é subclasse de Object, e com isso herda alguns métodos automaticamente. Um método muito interessante presente na classe Object é o equals. Suponha que haja duas instâncias de uma mesma classe e desejamos testar se elas contém a mesma informação. O operador == nos daria o valor true apenas se seus operandos forem precisamente o mesmo objeto. Porém, o operador equals nos diria quando os objetos contém o mesmo estado, através da comparação campo-a-campo. Por exemplo, eu e você podemos ter carro do mesmo modelo. Nesse caso meuCarro == seuCarro seria false pois embora nossos carros sejam do mesmo modelo, são carros diferentes. Entretanto, meuCarro.equals(seuCarro) poderia ser true se os atributos de ambos os carros fossem idênticos, por exemplo, mesmo ano, mesma cor, etc.

Um outro método interessante da classe Object é o método getClass, que retorna uma referência a um objeto contendo informações sobre a classe a que é aplicado. Isto será visto logo abaixo.

#### A classe **Class**

A classe Class contém informações que descrevem uma classe em Java. Toda classe em Java tem uma correspondente instância da classe Class. É possível obter informações contidas nessas instâncias por um dos seguintes meios:

* Usar o método getClass de um objeto para obter uma referência à respectiva instância da classe Class. Exemplo:
* Vértice v = new Vértice(1.0, 2.0);

Class cv = v.getClass();

* Usar o método estático forName de Class para obter uma instância de Class usando o nome da classe. Por exemplo:

Class cv = Class.forName("Vértice");

De posse de uma instância da classe Class, podemos obter informações interesantes  sobre a classe da qual ela provém. Por exemplo:

* Obter o nome da classe. Isto é muito útil quando lidamos com polimorfismo.
* Polígono p;
* ...
* p = new Retângulo( ... );
* ...

System.out.println("O polígono é um " + p.getClass().getName() );

deverá exibir na tela a mensagem:

O poligono é um Retângulo

* Obter o nome da superclasse imediatamente adjacente.
* Retângulo r;
* ...
* System.out.println("A classe parente do objeto é " +

   r.getClass().getSuperClass().getName() );

deverá exibir na tela a mensagem:

A classe parente do objeto é Polígono

Outra possibilididade interessante do uso da classe Class está na instanciação dinâmica de objetos:

     Polígono p;

     String nome;

     System.out.print("Qual o poligono que deseja criar?");

     System.out.flush();

     nome = System.in.read();

     p = (Polígono) Class.forName(nome).newInstance();

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), [seguinte](http://www.dm.ufscar.br/~waldeck/curso/java/part34.html)]

### Applets

As aplets são pequenos programas Java que podem ser inseridos dentro de páginas HTML. Com este recurso, uma página torna-se dinâmica, podendo interagir com o usuário que a consulte. Um applet pode ainda executar tarefas complexas, como realizar cálculos e apresentar gráficos, sons e imagens em movimento. O quadro abaixo apresenta uma applet com o clássico Jogo da Velha. Experimente clicar com o mouse em alguma de suas casas:

### Inserindo applets na página HTML

Para inserir uma applet numa página HTML, usamos a diretiva <applet>, a qual deve apresentar pelo menos três parametros: code, width e height. Assim, a especificação mais simples tem a forma:

<applet

    code=[java applet]

    width=[largura]

    height=[altura]>

onde [java applet] é o nome da classe principal (que identifica o programa) Java, e [largura] e [altura] indicam a largura e a altura em pixels da área dentro da página reservada para a apresentação da applet. Por exemplo, a applet do Jogo da Velha foi inserida nesta página através das diretivas:

<applet

    code=TicTacToe.class

    width=120

    height=120>

</applet>

Existem ainda, outros parâmetros adicionais que podem ser acrescentados opcionalmente:

* codebase: URL do diretório que contém as classes compiladas (bytecodes) do applet
* name: Um nome que identifica uma particular instância de um applet dentro de uma página HTML (útil para comunicação entre applets)
* align: especifica um alinhamento para a área da applet dentro da página. Análogo ao alinhamento de imagens
* alt: Texto a ser exibido na área da applet durante o carregamento, ou na impossibilidade de carregar a applet

Lembre-se que para atribuir a cada parâmetro um dado que não seja uma única palavra, é preciso encerrá-lo entre aspas.

### Passagem de parâmetros para a applet

Freqüentemente desejamos utilizar uma mesma applet em páginas diferentes, esperando, porém, que ela se comporte diferentemente em cada uma delas. Isto é possível através da passagem de parâmetros. Um parâmetro é especificado pela diretiva <param>, a qual deve aparecer dentro do escopo delimitado pelas diretivas <applet> e </applet>. A sintaxe dessa diretiva é a seguinte:

<param name=[nome] value=[valor]>

onde [nome] indica o nome do parâmetro, tal como é esperado pela applet, e [valor] representa a informação a ser passada. Como exemplo, vamos inserir abaixo uma applet chamado StringApplet que escreve um texto passado como parâmetro na área da applet:

<applet code=StringApplet.class width=360 height=40

 name="Applet de mansagem">

<param name=frase value="Alô, Pessoal!">

</applet>

isto nos dará a saída

### Navegador que não suporta Applets

No momento, nem todos os programas navegadores existentes dão suporte a applets Java. Muitos usuários ainda encontram-se utilizando versões antigas do Netscape em ambiente Windows 3.1, além de outros navegadores como o Mosaic, por exemplo, os quais foram concebidos antes do advento da linguagem Java. Tais navegadores simplesmente ignoram as diretivas <applet>, <param> e </applet>.

Como não temos meios para saber quando um usuário possui ou não um navegador adequado, necessitamos de uma maneira de polidamente advertí-lo sobre o uso de applets. Para isso, basta introduzir código HTML ordinário no escopo da diretiva <applet>. Os navegadores sem suporte para Java vão exibir esse conteúdo como se as diretivas de applets não estivessem lá. Já os navegadores com suporte a Java ignoram qualquer coisa que esteja nesse escopo, exceto, é claro, quando forem diretivas <param>.

Por exemplo:

<applet

    code=StringApplet.class

    width=360

    height=40

    align=middle>

<param name=frase value="Alô Pessoal!">

<p> Obs: esta página não poderá ser visualizada corretamente pelo

    seu navegador pois ele não é capaz de executar applets Java.

</applet>

### Escrevendo a primeira applet

Como qualquer programa Java, uma applet deve conter ao menos uma classe pública, e essa classe pública deve estender a classe Applet. A classe Applet faz parte do package applet que integra o *Java Developers Kit* (*JDK*) e define o comportamento básico que uma applet deve ter. Esta permite a codificação de uma applet mínimal em pouquíssimas linhas, como segue:

import java.applet.\*;

public class AloPessoal extends Applet {

}

A primeira linha deste código informa ao compilador Java que esta applet utilizará algumas ou todas as classes presentes no package applet.

A segunda linha declara a nova classe AloPessoal como derivada da classe Applet. Essa classe deve ser pública para que possamos acessá-la ao rodar a applet. Se não fosse pública, ainda assim poderia ser compilada, porém a applet não poderia ser executada. Na declaração de AloPessoal fazemos uso do conceito de *herança* da OOP por meio de derivação de classes. Esta derivação funciona da mesma maneira que vimos anteriormente no capítulo sobre [classes](http://www.dm.ufscar.br/~waldeck/curso/java/part27.html#herança). Agora, porém, a diferença é que a classe Applet foi escrita pelo pessoal da Sun e não por nós mesmos.

Podemos compilar e em seguida rodar essa applet. Para isso, basta criarmos uma página HTML com o seguinte código:

<html><head>

<title>Primeira Applet</title>

</head><body>

<applet code=AloPessoal.class width=320 height=80>

</applet>

</html>

Contudo, não seremos capazes de visualizar nada além de um retângulo cinza.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngDurante a fase de testes não é conveniente utilizarmos um programa navegador para executar uma applet. Isto por causa do mecanismo de cache presente na maioria dos navegadores que lhes permite uma melhor performance durante a carga das páginas. Esse mecanismo freqüêntemente se atrapalha ao lidar com versões diferentes de uma mesma applet. Existe, porém, um programa no JDK chamado appletviewer. O appletviewer é uma maneira muito conveniente de se testar e validar uma única applet antes que ela seja introduzida em uma página HTML. Para executar uma applet com o appletviewer, basta chamar o programa passando-lhe o nome de uma página HTML contendo apenas as diretivas necessárias à applet.

Vamos acrescentar a essa applet código necessário para que ela escreva o texto "Alô Pessoal!" em sua janela:

import java.applet.\*;

import java.awt.\*;

public class AloPessoal extends Applet {

   public void paint(Graphics g) {

      g.drawString("Alô Pessoal!", 100, 20);

   }

}

O código acrescentado neste exemplo ficará mais claro com a discussão que faremos no tópico seguinte.

### Applet que recebem parâmetros

Agora vamos desenvolvelr um applet que reconhece os parâmetros passados no HTML. Para isso, vamos implementar o applet Vamos implementar o StringApplet discutido na seçào de [passagem de parâmetros para applet](http://www.dm.ufscar.br/~waldeck/curso/java/part33.html#passagem_de_par�metros).   
Todo parâmetro passado de página Web para applet possui um nome e valor literal. Para obter o valor do parâmetro, usa-se o método getParam. O StringApplet recebe o parâmetro com nome frase e desenha na janela.

import java.applet.\*;

import java.awt.\*;

public class StringApplet extends Applet {

   String msg;

   public void paint(Graphics g) {

      msg = getParam("frase");

      g.drawString(msg, 100, 20);

   }

}

Agora, se quiser aperfeiçoar mais um pouco e escrever a mensagem, iniciando na coordenada indicada? É simples. Basta definir dois variáveis para designar a coordenada x e y. Porém, os parâmetros passados da página para applet sempre é um string. Para converter string para inteiros, usa-se o método parseInt().

import java.applet.\*;

import java.awt.\*;

public class Stringapplet2 extends Applet {

   String msg, int x,y;

   public void paint(Graphics g) {

      msg = getParam("frase");

      x = Integer.parseInt(getParam("xcoord");

      Y = Integer.parseInt(getParam("ycoord");

      g.drawString(msg, x, y);

   }

}

As coordenadas x e y do ponto inicial da frase foi considerado de "xcoord" e "ycoord" respectivamente. Para escrever "Alo, pessoal!" começando de (10,20), basta inserir o código

<applet code=StringApplet.class width=360 height=40   
 name="Applet de mansagem">   
<param name=frase value="Alô, Pessoal!">   
<param name="xcoord" value = 10>   
<param name=ycoord value=20>   
</applet>

na página HTML.

### Desenhando na Janela

Embora a maioria das applets que escrevemos sejam razoavelmente simples, há certas coisas de que precisamos saber para podermos escrever nossas próprias applets. O espaço de uma página HTML reservado à uma applet, isto é, sua janela, é uma área gráfica. Isto significa que precisamos saber como desenhar nessa janela. Para isso, devemos usar a classe Java chamada Graphics. Apenas isso, porém, não é o suficiente. Precisamos aprender como utilizar os diversos tipos de controles existentes, como botões, caixas de texto, listas de rolagem, etc, e como dispor esses controles na janela segundo uma ordem lógica.

Uma outra coisa muito importante a saber é como tratar eventos. A movimentação e o clicar do mouse, o pressionar de uma tecla, etc, são eventos que ocorrem durante a execução de um programa. Tais eventos produzem mensagens que são enviadas à applet. Como resultado da recepção dessas mensagens, ocorre dentro da applet a chamada de métodos. A applet deve prover métodos para tratar de todos os eventos que sejam necessários à realização de sua tarefa.

Responder aos movimentos do mouse não é coisa que se tenha que fazer com freqüência. Entretanto é interessante saber como fazê-lo. Tal tratamento seria indispensável se estivéssemos escrevendo um jogo, por exemplo, ou um programa gráfico que permitisse desenhar na janela através do mouse. Nosso próximo exemplo será justamente um programa que permite fazer isso. Ao executar essa a Applet Davinci no appletviewer vemos uma janela em branco. Clicando com o mouse e em seguida arrastando para alguma direção, veremos um traço preto acompanhando o apontador onde quer que o arrastemos. Apesar de ser um programa muito simples, nos dá uma idéia clara sobre como utilizar o mouse e o dispositivo gráfico para realizar tarefas similares.

import java.awt.\*;

import java.applet.\*;

public class Davinci extends Applet

{

    Point pontoInicial;

    Point pontos[];

    int numpontos;

    boolean desenhando;

    public void init()

    {

        pontoInicial = new Point(0, 0);

        pontos = new Point[1000];

        numpontos = 0;

        desenhando = false;

    }

    public void paint(Graphics g)

    {

        int antX = pontoInicial.x;

        int antY = pontoInicial.y;

        for (int i=0; i<numpontos; i++)

        {

            g.drawLine(antX, antY, pontos[i].x, pontos[i].y);

            antX = pontos[i].x;

            antY = pontos[i].y;

        }

    }

    public boolean mouseDown(Event evt, int x, int y)

    {

        if (!desenhando)

        {

            desenhando = true;

            pontoInicial.x = x;

            pontoInicial.y = y;

        }

        else

            desenhando = false;

        return true;

    }

    public boolean mouseMove(Event evt, int x, int y)

    {

        if ((desenhando) && (numpontos < 1000))

        {

            pontos[numpontos] = new Point(x, y);

            ++numpontos;

            repaint();

        }

        return true;

    }

}

Veja como essa applet responde aos cliques do mouse:

#### Compreendendo a applet Davinci

Vamos agora estudar detalhadamente a applet Davinci. As duas primeiras linhas informam ao compilador que estaremos usando classes dos packages awt e applet:

import java.awt.\*;

import java.applet.\*;

O package awt (Abstract Window Toolkit) contém dezenas de classes para desenhar em janelas. Estas classes dão-nos acesso a métodos para desenhar na tela, responder a eventos, manipular controles como botões e menus, etc. O package applet contém classes necessárias para se construir um applet báciso.

A próxima linha do programa declara a classe Davinci como subclasse da classe Applet.

public class Davinci extends Applet

Em seguida temos a declaração dos dados de que a applet necessitará.

    Point pontoInicial;

    Point pontos[];

    int numpontos;

    boolean desenhando;

A classe Point está definida no package awt e representa uma estrutura de dados contendo um par de números x e y, isto é, as coordenadas de um ponto. O inteiro numpontos dá o número de pontos armazenados no vetor pontos e o booleano desenhando indica que estamos no modo de desenho.

#### O método init()

O método init está definido na classe Applet. Estamos, portanto, fazendo uma sobreposição desse método para prover informações iniciais adicionais para a applet. Como veremos adiante, o método init é um dos métodos que compões o ciclo normal de vida de uma applet. Esse método é chamado pelo programa navegador imediatamente após a applet ter sido carregada. No caso de nossa applet, a primeira coisa a fazer é criar um novo objeto Point para conter as coordenadas do ponto inicial do desenho:

pontoInicial = new Point(0, 0);

Aqui o operador new cria um novo objeto da classe Point e chama seu construtor com os valores (0, 0), os quais serão atribuídos aos campos x e y respectivamente. Depois, precisamos criar um vetor de objetos da classe Point para conter as coordenadas dos pontos que serão traçados:

pontos = new Point[1000];

Nas próximas duas linhas, inicializamos as duas variáveis restantes:

numpontos = 0;

desenhando = false;

A primeira delas indica a quantidade total de pontos presentes no vetor pontos, que no início é zero. A segunda indica que não estamos no modo de desenho, isto é, passear com o mouse por cima da janela da applet não provocará o traçado do desenho.

#### O método mouseDown()

Como qualquer outro programa que trabalhe com janelas, as applets devem responder a eventos do mouse, como os de movimento e de clicagem, por exemplo. O método mouseDown é chamado sempre que o usuário fizer um clique com o mouse sobre a área da janela. Os argumentos desse método são um objeto da classe Event, a qual veremos mais adiante, e as coordenadas x e y do ponto da janela em que ocorreu o clique. Em nossa applet, na ocorrência desse evento, devemos verificar em que modo estamos trabalhando, se de desenho ou não, pois isso afetará a maneira de tratar os eventos subseqüêntes:

if (!desenhando)

Se não estivermos desenhando, então o clicar do mouse indica que o usuário deseja iniciar um novo traçado. Devemos então alternar para o modo de desenho, guardando as coordenadas x e y do ponto de clicagem como as coordenadas iniciais do novo traçado:

desenhando = true;

pontoInicial.x = x;

pontoInicial.y = y;

Se já estivermos no modo de desenho, tudo o que precisamos fazer é alternar para o modo normal:

else

   desenhando = false;

#### método mouseMove()

Este método é chamado sempre que ocorrer mudança de posição do mouse dentro da área do applet. o seu argumento é objeto Event e coordenadas x, y do mouse . Este método foi redefinido para acrescentar um ponto com coordenada do mouse a mais no vetor de pontos. Note que testa se o vetor já está cheio.   
Após acrescentar pontos no vetor, chama o método repaint().

### Os quatro estágios de execução de uma applet

Além do estágio inicial, caracterizado pela chamada o método init, uma applet tem mais três outros: partida, parada e destruição.

O estágio de partida ocorre quando se inicia a execução da applet, imediatamente após o estágio inicial, ou quando a applet for reiniciada. Isso geralmente acontece quando o usuário pressiona o botão "voltar" do navegador após ter visualizado uma outra página. Diferente do estágio inicial, o estágio de partida pode ocorrer múltiplas vezes durante uma sessão. Para interceptar esse estado, sobrepomos o método start.

O estágio de parada ocorre quando uma applet se torna invisível na tela. Isso ocorre quando o usuário abandona a página em que se encontra a applet para consultar uma outra. Isso não significa que a applet deve parar de executar; ela poderá continuar realizando alguma tarefa não aparente. Se este for o caso, podemos então sobrepor o método stop.

Finalmente o estágio de destruição é aquele em que a applet está sendo removida da memória e todos os recursos por ela alocados são devolvidos ao sistema operacional. Assim como estágio de inicial, o estágio de destruição ocorre apenas uma vez, mediante a chamada do método destroy.

### Programação Orientada a Eventos

O desenvolvimento de applets baseia-se no modelo de execução de programa conhecido como *execução orientada a eventos*. Este paradigma de programação difere significativamente do tradicional na forma como se dá o controle de fluxo da execução.

Na programação tradicional, quando o programa precisa ler um dado do teclado, por exemplo, há a chamada para uma função do sistema que realiza a leitura do teclado. Essa função fica continuamente monitorando o teclado para poder capturar o valor da tecla pressionada. Assim, a execução do programa fica suspensa até o momento em que o usuário resolve prosseguir, pressionando uma tecla.

Por outro lado, na programação orientada a eventos, quando o programa necessita de um dado ele simplesmente avisa ao usuário sobre essa necessidade, geralmente imprimindo alguma mensagem na tela, e em seguida continua o processamento realizando alguma outra tarefa, se for o caso. Quando o usuário pressiona uma tecla, o sistema operacional se encarrega de avisar ao programa sobre a ocorrência desse "evento" e dar a ele a chance de realizar o que for preciso com a informação fornecida pelo usuário. Há, contudo, outros tipos de eventos. Um evento pode ser o clicar ou mover do mouse, entrada de texto, necessidade de atualizar uma janela, disparo de um temporizador, etc. Processar eventos provenientes do mouse, por exemplo, é uma grande dificuldade para a programação tradicional, pois geralmente o programa não tem como prever sobre qual "botão virtual" o usuário vai clicar. No caso da applet de desenho, simplesmente indicamos o que fazer para cada alteração do mouse.

Os eventos principais das applets são:

* mouse
* teclado
* botões
* caixa de diálogo, etc

A seguir, vamos tratar de eventos provenientes do teclado e do mouse.

### Teclado

Existem apenas dois eventos que podem provir do teclado: keyDown e keyUp.

public boolean keyDown(Event evento, int keyCode)

Este método é chamado quando alguma tecla for pressionada. keyCode é um número que dá um código correspondente à tecla precionada. Como existem teclados sem padronização, existem determinados códigos especiais para algumas designar certas teclas:

* Event.F1 a Event.F12: Tecla de função F1-F12
* Event.LEFT: seta para esquerda
* Event.RIGHT: seta para direita
* Event.UP: seta para cima
* Event.DOWN: seta para baixo
* Event.END: tecla End
* Event.HOME: tecla Home
* Event.PGDN: tecla Page Down
* Event.PGUP: tecla Page Up

A informação de que a tecla *shift*, *control* e *alt* (ou *meta*) foi pressionada é obtida do parâmetro event recebido, usando método correspondente. Todos eles retornam um dado booleano indicando a situação da respectiva tecla. Suponha que o nome do parâmetro de evento (do tipo Event) é evento. Então temos

* evento.shiftDown() retorna true, se a tecla *Shift* estiver pressionada no momento da ocorrência do evento
* evento.ctrlDown() retorna true se a tecla *Ctrl* estiver pressionada
* evento.metaDown() retorna true se a tecla *Alt* (*meta*) estiver pressionada

public boolean keyUp(Event event, int keyCode)

Este método é chamado quando a tecla pressionada é liberada pelo usuário. Os parâmetros recebidos são os mesmo do método keyDown.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngNão se esqueça de que ocorrem dois eventos ao pressionar uma tecla: keyDown e keyUp. Não é, porém, obrigatório tratar ambos os eventos. Esses métodos já estão codificados na classe Applet provendo um tratamento padrão para os respectivos eventos.

### Mouse

Os métodos associados eventos provenientes do mouse recebem dois parâmetros: evento do tipo Event e coordenadas x e y da posição do ponteiro mouse (do tipo inteiro) relativos às coordenadas d janela atual. Estes já foram bem explicados acima.

Bem, além do método mouseDown e mouseMove, existe ainda alguns métodos, recebendo mesmo parâmetro que estes:

**public boolean mouseUp(Event event, int x, int y)**

Este método é chamado quando botão do mouse que estava precionado for solto.

**public boolean mouseDrag(Event event, int x, int y)**

Este método é chamado, quando o usuário arrastar o mouse (precionar o botão, mudar de posição sem soltar o botão e em seguida, soltar o botão).

**public boolean mouseEnter(Event event, int x, int y)  
public boolean mouseExit(Event event, int x, int y)**

São chamadas respectivamente quando mouse entra e sai do objeto em que o método pertence. Podemos usar este método por exemplo, para acender o botão em que o mouse está, antes msmo do clique de mouse.

http://www.dm.ufscar.br/~waldeck/curso/java/coffee.pngNo evento do mouse, não há como detectar o botão (se é direito ou esquerdo) do mouse precionado. Isto deve ao fato do mouse dos computadores da Apple possuir apenas um botão, e Java, por poder ser executado em qualquer plataforma, assume que o mouse possui apenas um botão.

**Projeto 01**

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part33.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), [seguinte](http://www.dm.ufscar.br/~waldeck/curso/java/part36.html)]

Neste projeto, vamos desenvolver applet na qual possa desenhar e visualizar algumas figuras geométricas simples, aperfeiçoando assim, a técnica de programação.

**Primeira etapa: Classe de figuras**

Vamos criar um applet na qual, desenha um quadrado, circulo e triangulo. Para isso, vamos implementar classes classe base denominado de figuras e usar polimorfísmo para poder desenhar e manipular figuras de forma única.   
A classe figura contém um ponto que serve como referência para posição da figura, uma cor e dois métodos virtuais:

* pertence: que testa, se o ponto está dentro da figura. Este método é essencial para testar se o mouse foi clicado sobre a figura.
* desenha:: este método desenha a figura preenchido.
* Vários construtores, inclusive uma que cria figura igual ao figura passado nele. Este método é similar a clone(), e permite criar cópias da classe definido.

class figura {

    Point P; // ponto de referência da posição do objeto

    public Color cor; // cor da figura

    public void figura() { P = new Point(0,0); cor = new Color(0,0,0);} // construtor default

    public void figura(int x, int y) { P = new Point(x,y);}

    public void figura(int x, int y, Color cor) { P = new Point(x,y);

         this.cor = new Color(cor.getRed(), cor.getBlue(), cor.getGreen()); }

public void figura(figura fig) {

figura(fig.p.x, fig.p.y, fig.cor); }

    // métodos virtuais

    public bool pertence(Point P);

    public void desenha(Graphics g); // desenha a figura preenchida na saída Graphics

}

Uma classe derivada da classe figura devem implementar o método pertence e desenha. Por exemplo, uma classe quadrado que possui dois pontos de referência (canto superior esquerdo e canto inferior direito) pode ser implementado como sendo

class retangulo extends figura {

    // P definido na super classe e o canto superior esquerdo

   Point Q; // canto inferior direito

    public void retangulo() { super(); Q = new Point(0,0); } // construtor default

    public void retangulo(int x1, int y1, int x2, int y2) { super(x1,y1); Q = new Point(x2,y2); }

public void retangulo(int x1, int y2, int x2, int y2, color cor) {

super(x1,y1,cor); Q = new Point(x2,y2); }

public void retangulo(retangulo retang) {

super(retang); Q = new Point(x2,y2); }

    // métodos virtuais

    public bool pertence(Point P) {

      return ( (this.P.x<=P.x)&&(P.x<=Q.x) && (this.P.y<=P.y)&&(P.y<=Q.y) );

   }

   public void desenha(Graphics g) { // desenha a figura preenchido saída Graphics

     int coordx[]={P.x, P.x, Q.x, Q.x};

     int coordy[]={P.y, Q.y, Q.y, P.y};

g.setcolor(cor); // seta a cor

     Polygon  retang=new Polygon(coordx, coordy, 4); // retângulo é poligono de quatro vértices.

     g.fillPolygon(retang);

 }

}

A classe Polygon e a classe de polígono e, é criado, passando dois vetores de interiros (lista de coordenadas X e Y respectivamente) e um inteiro, indicando o número de vértices. O método fillPolygon da classe Graphics desenha o poligono preenchido.

**Exercício 1:**   
Agora implemente uma classe extendido de figura, denominado de circulo. Lembre-se que:

1. O círculo pode ser representado pelo centro e raio.
2. O método  
   public abstract void fillOval(int x, int y, int width, int height)   
   desenha elipse preenchido, com centro em (x,y) com raios width e height nos eixos X e Y, respectivamente. Não esqueça de ajustar a cor.
3. Um ponto X pertence a um círculo de raio r com centro em P se e somente se <X-P> <= r\*r

A classe deve ter construtores similares a da classe retangulo.

**Exercício 2:**   
Implemente também, a classe triangulo. lembre-se que:

1. O triângulo pode ser representado por três pontos.
2. O método  
   bool inside(int x, int y)  
   da classe Polygon testa se o ponto (x,y) pertence ao poligono.

Esta classe será implementado de forma quase igual ao retangulo.

**Exercício 3:**Crie um applet para testar as três figuras acima e verifique se pode criar e desenhar corretamente.

**Segunda etapa; classe de desenhos**

Agora, vamos criar classe denominado de desenho que compõe de um vetor de figuras, e métodos de manipulação.

**Campo:**Inicialmente, queremos que seja capaz de armazenar figuras com rótulo (nome) para facilitar a manipulação. Assim, esta classe deve possuir dois vetores: um vetor de classe figura e outro de String para guardar o nome correspondente. Claro, não devemos esquecer do número de elementos.

Métodos:

* Esta classe deve ser capaz de inserir ou deletar figuras do vetor de figuras, referênciando pelo nome.
* Deve ser capaz de desenhar todas figuras em ordem, sobrepondo-os segundo a prioridade.
* dado ponto (x,y), deve identificar qual das figuras desenhadas contém este ponto. isto permite selecionar figura pelo mouse.
* permite mudar a prioridade. Dado um nome, deve ser capaz de colocar por último, permitindo assim, que esta figura seja desenhado por último.

class desenho {

figura figs[]; // lista de desenhos

String nomes[]; // lista dos nomes das figuras

int numfigs; // número de figuras

int maxfigs; // número máximo de figuras

// construtor

public desenho(){

figs=new figura[50];

nomes=new String[50];

numfigs=0;

maxfigs=50;

}

public desenho(int numfigs) {

figs=new figura[numfigs];

nomes=new String[numfigs];

this.numfigs=0;

maxfigs=numfigs;

}

// métodos auxiliares.

private int procura(String nomefig) {

// retorna o índice da figura.

// caso não for encontrado, retorna numfigs

int i=0;

for(i=0;i<numfigs;i++)

if(nomes[i].equal(nomefig)) break;

return i;

}

// insere figura.

public bool adiciona(String nome, figura fig) {

if(numfigs<maxfigs) {

figs[numfigs]=new figura(fig);

nomes[numfigs]=new String(nome);

numfigs++;

return true;

}

return false;

}

// elimina figura

public bool elimina(String nome) {

int i = procura(nome);

if(i<numfigs) {

for(;i<numfigs-1;i++) {

figs[i]=figs[i+1];

nomes[i]=nomes[i+1];

numfigs--;

}

return true;

}

return false;

}

// desenha figura

public void desenha(Graphics g) {

int i;

for(i=0;i<numfigs;i++)

figs[i].desenha(g);

}

// encontra a primeira figura que contém (x,y)

// retorna "" caso não existir

public String pertence(int x, int y) {

int i;

for(i=numfigs-1;i>=0;i--)

if(figs[i].pertence(x,y))

return nomes[i];

return new String("");

 }

// faz com que a figura indicada tenha primeira prioridade

// fique por cima

public void prioridade(String nome) {

int i=procura(nome);

figura tempfig;

String tempnome;

if(i<numfigs) {

temp = figs[i];

tempnome = nomes[i];

for(;i<numfigs-1;i++) {

figs[i] = figs[i+1];

nomes[i] = nomes[i+1];

}

figs[i]=tempfig;

nomes[i]=tempnome;

}

}

// testa, se a figura se encontra

public bool pertence(String nome) {

return (procura(nome)<numfigs);

}

// este método retorna a figura com nome especificado,

// caso não encontrar, gera uma excessão

// "Ele não tira cópia!"

public figura pegafigura(String nome) {

int i = procura(nome);

if(i<numfigs)

return figs[i];

else {

return

trows NoSuchElementException; // definido no java.util

// coloque import java.util.\*; no começo do arquivo.

}

}

**Exercício 1:**Analise a classe acima com cuidado e em seguida, faça um applet para testar.

**Exercício 2:**Faça com que o applet identifique a figura clicada pelo mouse e coloca na primeira prioridade (desenhar por último). Para isso, defina o método mouseDown();

[[anterior](http://www.dm.ufscar.br/~waldeck/curso/java/part34.html), [índice](http://www.dm.ufscar.br/~waldeck/curso/java/), seguinte]

### AWT

AWT (Abstract Window Toolkit) é biblioteca da classe java que implementa interface do applet através de janelas. Note que, o applet pode ser implementado para ser executado fora da página web e por isso, AWT pode ser usado para implementar interface de aplicativos escritos em java. Nos veremos ainda, como desenvolver applets que podem ser executador fora da página.

### Introdução ao AWT

AWT é um conjunto de recursos gráficos comumente conhecidos pelos sistemas de interface usando janelas. Nos podemos criar janelas, botões, janela de diálogos, etc.   
Estas ferramentas possui uma iherarquia, que será explicado mais em diante.   
Por enquanto, apenas aprenderemos como usar o AWT que está baseado no interface orientado a enventos.   
Programação orientado a eventos é uma técnica de programação na qual, costruimos o programa através de regras (métodos) a ser executados cada vez que o evento diferente ocorrer. Por exemplo, precionar teclado, mover mouse, clicar no botão, receber dados pela rede, todos são eventos. Cada vez que um desses eventos ocorrer, o método correspondente é ativado.   
O elemento de AWT são na maioria, componentes. isto quer dizer que eles podem ser adicionados numa janela (frame) ou painel (panel). Para começar, precisamos de uma janela ou uma área em branco para poder colocar os botões e menus.   
O applet ja possui sua área própria que é um retângulo sem nada, denominado de **panel**. Então, ele pode começar a colocar (adicionar) botões e outros componentes. Se quiser sub-dividir, basta acrescentar panel's.   
Quando ocorrer qualquer evento no componente do AWT, o método denominado de **action** é chamado. O que nos precisamos fazer é implementar o método action para identificar o evento ocorrido e manipular corretamente.   
Em vez de ficar discutindo monte de coisas, vamos para parte prática. Suponhe que queremos adicionar botões no panel do applet (na parte de [escrevendo o applet](http://www.dm.ufscar.br/~waldeck/curso/java/part24.html), a área de trabalho de applet dentro da página HTML foi referenciado como sendo janelas, mas referência correta é painel).

### Botão

Botão é uma classe denominado de Button e possui um nome. O evento do botão é cliqui do mouse sobre ele, ou seja, quando mouse é clicado sobre ele, chama o método **action**.   
Ele possui três métodos principais:

public Button() // construtor. Cria o botão   
public Button(String label) // Cria o botão, já dando nome para ele.   
public void setLabel(String label) // Atribui nome para botão   
public String getLabel() // Retorna o nome do botão

Para inserir algo no componente de AWT, utiliza o método add associado a ele. Se o add for chamado sem especificação do objeto, é considerado objeto corrente. Agora, vamos construir um applet simples que altera a cor do panel do applet, clicando sobre botão.

import java.awt.\*;

import java.applet.\*;

public class Botao extends Applet {

  public void init(){

    add(new Button("Azul"));

    add(new Button("Vermelho"));

  }

  public void boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

      String label = (String)quem;

      if(label == "Azul")

         setBackground(Color.blue);

      else if (label == "Vermelho")

         setBackground(Color.red));

      else

         return false;

      repaint();

      return true; // não chama mais nenhum action seguinte

    }

    else

      return false; // evento não foi de botão. Próximo action é chamado.

  }

}

Vamos esclarecer as coisas. Primeiro, o método init() que é executado quando o applet inicia, adiciona dois botões no panel da applet, com nome "Azul"  e "Vermelho" respectivamente.   
Quando o botão for clicado, ocorre o evento no botão, que não possui regras (métodos) associados. Quando o objeto não possui método para cuidar de evento (caso do botão acima), o evento será passado para a elemento que contém ele. No caso do exemplo acima, o botão foi adicionado no panel da applet e logo, o action da applet é chamado.   
Os métodos de tratamento do evento é:   
public boolean action(Event evento, Object quem)   
onde o parâmetro evento contém informações sobre evento e quem é o objeto na qual o evento ocorreu.   
Note que, o evento de mouse e teclado discutido no escrevendo o applet é um evento específico do panel e não inclui nenhum evento tais como clicar botão, selecionar ítem, entrar com texto, etc. Todos esses eventos serão tratados pelo método action.   
Primeiro, o action verifica se o evento foi gerado pelo botão. Para isso, basta verificar se o evento.target é uma instância de Button. isto é feito pelo (evento.target instanceof Button). Caso for botão, podemos tomar decisão de acordo com o botão clicado. Para saber qual dos botões foi precionado, converte o objeto quem em String. note que quem é o objeto (botão) que foi clicado. Quando converte para nome, será nome do botão que é armazenado no variável label e é verificado se é "azul" ou "vermelho". Em seguida, muda as cores do fundo. A classe Color possui várias definições de cores usuais, assim como permite criar cores a partir do modelo RGB ou HSI.   
Agora, note que retorna false quando o evento não é do botão ou não for botão específico. Quando o método que cuida do evento retornar true, o AWT assume que o evento foi devidamente  tratado. quando o false é retornado, o método de tratamento do evento (mouse, teclado ou action) da classe que adicionou ele será chamado. Por exemplo, se o Applet fizer parte de algum outro componente (alguma janela, por exemplo), se action do applet retornar false, o action da janela é chamado. se o action do applet retornar true, o método da janela não será chamado. isto fica mais evidênte na medida que for acostumando com o interface através de AWT e começar adicionar componentes um dentro do outro.

### Rótulo:

Um rótulo é denominado de label. A utilidade dele é apresentar textos dentro do panel ou frame.   
A classe Label possui três construtores:   
public Label()   
public Label(String labeltext)   
public label(String Labeltext,. int align);   
O Label pode ser construido com ou sem o texto e alinhamento especificado. Para alinhamento, existe três tipos definidos na classe label: RIGHT (direita), LEFT (esquerda), CENTER (centrado). Não esqueça que, para acessar o valor definido na classe, devemos colocar o nome da classe antes do valor, tais como Labal.LEFT.   
Possui dois métodos:   
public void setText(String texto);   
public String getText();   
Estes métodos atribuem (ou modificam) e retornam respectivamente, o valor a ser mostrado no Label.

import java.awt.\*;

import java.applet.\*;

public class Rotulo extends Applet {

  Label texto = Label("Alo, pessoal!", Label.LEFT);

  boolean original = true;

  public void init(){

    add(new Button("mudar o rotulo"));

  }

  public void boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

      if(original)

         texto.setText("Sou applet de rótulo");

      else

         texto.setText("Alo, pessoa1");

      original = !original;

      repaint();

      return true;

    }

    else

      return false; // evento não foi de botão. Próximo action é chamado.

  }

}

Note que não verificamos o nome do botão. Isto pode ser feito, porque tem apenas um botão e logo, se algum botão for clicado, deve ser ele.

### Caixa de checagem

Caixa de ticagem é uma espécie de label, com opção de ligar/desliga. Ele apresenta um quadradinho seguido de label, e o usuário pode ligar ou desligar esta opção. Ele é denominado de CheckBox e possui três construtores:   
public Checkboxl()   
public Checkbox(String labeltext)   
public Checkbox(String Labeltext, CheckboxGroup g, boolean ini)   
O terceiro construtor parece estranho, mas isto é devido ao fato de Checkbox é usado também para criar botões de rádio.   
O primeiro e segundo construtor tem nada a esplicar. O terceiro construtor é usado, quando deseja criar ChekBox já ligado ou desligao pelo default. O segundo parâmetro CheckboxGroup, por enquanto, esquece. Passe o new Checkboxgoup() no lugar dele que instanciará um novo checkboxgroup.   
existe o método denominado   
public boolean getState()   
que retorna true se o CheckBox estiver ativado e false caso contrário.   
Cada vez que o usuário clicar sobre checkbox para ligar ou desligar, o evento ocorre e action é chamado.

import java.awt.\*;

import java.applet.\*;

public class MyCheckBox extends Applet {

  Checkbox Check1 = new CheckBox("gosto de java", new CheckboxGroup(), true),

         , Check2 = new CheckBox("iniciante em java", new CheckboxGroup(), true);

  public void init() {

    add(Check1);

    add(Check2);

  }

  public void boolean action(Event evento, Object quem) {

    if(evento.target instanceof CheckBox) {

      CheckBox checkboxAtual = (CheckBox)evento.target;

      if(CheckboxAtual.getLabel() == "gosto de java") { // primeiro Checkbox

         if(CheckboxAtual.getState()) { // ligado:

           // O que vai fazer, se ligar

         }

         else {

           // oque fazer, caso contrário.

         }

         return true;

       }

       else if(CheckboxAtual.getLabel() == "iniciante em java") {

         if(CheckboxAtual.getState()) {

             // o que fazer

          }

          else {

            // o que fazer

          }

          return true;

      }

      return false;

  }

}

No exemplo acima, armazenamos o evento.target no variável. Sempre que faz uso repetido, esta técnica facilita o processo.   
Normalmente, não implementamos o método action para tratar de CheckBox, toda vez que clicar sobre ele. Programação correta é incluir um botão de confirmação e checar o estado dos CheckBox somente quando o botão for clicado. Portanto, o exemplo acima serve apenas como exemplo para ilustrar o fato de que o checkbox gera evento e é possível tratá-lo.   
Verifique no exemplo abaixo, como tudo fica mais simples, se utilizar um botão de confirmação (esta é a progrmação usual de CheckBox).

import java.awt.\*;

import java.applet.\*;

public class MyCheckBox extends Applet {

  Checkbox Check1 = new CheckBox("gosto de java", new CheckboxGroup(), true),

         , Check2 = new Checkbox("iniciante em java", new CheckboxGroup(), true;

  public void init() {

    add(Check1);

    add(Check2);

    add(new Button("OK"));

  }

  public void boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) { // único botão é de confirmação

      if(Check1.getState()) { // ligado: ele gosta de java

        // O que vai fazer, se ligar

      }

      else {  // desligado: não gosta de java

        // oque fazer, caso contrário.

      }

      if(Check2.getState()) { // ligado: iniciante em java

        // o que fazer

      }

      else { // desligado: não é iniciante em java

       // o que fazer

      }

      return true;

    }

    return false;

  }

}

### Botão de Rádio.

Botão de rádio, denominado de Radio Button é um conjunto de checkbox e é implementado, usando o CheckBox. Neste conjunto, apenas um ítem estará ligado. É uma espécie de seleção de opção. Ao clicar sobre um deles, desativa automaticamente o outro. Para cliar o botão de rádio, criamos um objeto do tipo CheckboxGroup e ir adicionando Checkbox nele, usando o terceiro construtor do checkbox:   
public Checkbox(String texto, CheckboxGroup g, boolean ini)   
a classe CheckboxGroup possui um me'todo   
public Checkbox getCurrent()   
que retorna o checkBox escolhido atualmente (ítem selecionado).   
Não esqueça que, bom estilo de programação para usar Checkbox ou Radio Button é criar um botão extra de confirmação e checar o estado de Checkbox ou RadioButton somente quando for clicado sobre botão de confirmação.

import java.awt.\*;

import java.applet.\*;

public class MyCheckBox extends Applet {

  Label texto = new Label();

  CheckboxGroup Escolha = new CheckboxGroup();

  public void init() {

    add(new CheckBox("Linguagem favorito é Java, Escolha, true));

    add(new CheckBox("Linguagem favorito é C++, Escolha, false));

    add(new CheckBox("Linguagem favorito é Pascal, Escolha, false));

    add(new CheckBox("Linguagem favorito é Basic, Escolha, false));

    add(new CheckBox("Linguagem favorito é Fortran, Escolha, false));

    add(new Button("Confirma"));

    add(new Button("Cancela"));

    add(texto);

  }

  public boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

       if((String)quem == "Confirma") {

         CheckBox atual = Escolha.getCurrent();

         texto.setText(atual.getLabel());

         repaint();

         return true;

       }

     }

     return false;

  }

}

### Caixa de Escolha

A caixa de escolha é uma caixa que permite escolher ítens entre vários. Ele exibe o ítem selecionado na caixa até que altere a seleção. neste sentido, é semelhante ao botào de rádio.   
Ele possui como método básico

public suncronized void additem(String intem)

      throws NullPointerException

public void select(String intem)

public String getSelecteditem()

que servem respectivamente para adicionar ítem, escolher a seleção atual e obter o ítem selecionado atualmente.   
Existem outros métodos associados a ele, mas os três métodos acima já são suficientes para aplicação em geral. caso deseja, consulte o livro sobre outros métodos.   
Assim como CheckBox e Radio Buttom, Choice gera evento quando uma seleção é realizado. Na aplicação em geral, cada vez que efetua a escolha diferente, é efetuado a operação correspondênte, mas podemos usar botão extra como no caso de Checkbox e Radio Buttom.   
Veremos um exemplo de caixa de seleção para escolher a cor do fundo do panel de applet:

import java.awt.\*;

import java.applet.\*;

public class MyChoice extends Applet {

  Choice cor = new Choice();

  public void init() {

    cor.addItem("Vermelho");

    cor.addItem("Azul");

    cor.addItem("Verde");

    cor.additem("Amarelo");

    cor.addItem("Branco");

    cor.select("Branco");

    setBackground(Color.White);

    add(cor);

  }

  public boolean action(Event evento, Object quem) {

    if(evento.target instanceof Choice) {

       if(cor.getSelecteditem() == "Vermelho")

          setBackground(Color.Red);

       else if(cor.getSelecteditem() == "Azul")

          setBackground(Color.Blue);

       else if(cor.getSelecteditem() == "Verde")

          setBackground(Color.Green);

       else if(cor.getSelecteditem() == "Amarelo")

          setBackground(Color.yellow);

       else if(cor.getSelecteditem() == "Branco")

          setBackground(Color.White);

       else

          return false;

       return true;

     }

     return false;

    }

  }

}

### Lista (de rolagem)

A lista de rolagem é quase igual à caixa de seleção. Ele exibe os ítens para ser selecionado, porém, pode conter número grande de dados e pode rolar a lista atrvés da barra de rolagem ao lado dele. Os ítens serão mostrados numa área designada para ele. Na caixa de seleção, apenas um ítem pode ser selecionado, no entanto, na lista, pode ser configurado para selecionar apenas um ítem ou permitir multiplas seleções.   
Construtores são:   
public List()   
public list(int linhas, boolean MultiplaEscolha);   
No primeiro construtor, a list é da escolha simplas. No segundo construtor, podemos indicar o número de ítens que será exibido por vez e dizer, se a escolha é apenas um ítem ou multipla escolha.   
Para adicionar, ítens, usa-se o método:   
public syncronized void addItem("String item);   
 Para saber os ítens selecionados, podemos usar os métodos:   
public syncronized String getSelectedItem();   
public syncronized String getSelectedItens();   
Respectivamente para lista de escolha simples e escolha multipal.   
Uma observação importante é que, não chama o método action quando selecionamos elementos da lista. Portanto, devemos sempre criar um botão de confirmação. Mas caso queira manipular evento quando a seleção ocorre (por exemplo, escolha de fonte no editor), podemos usar o método   
public boolean handleEvent(Event event);   
Veremos um exemplo simples:

import java.awt.\*;

import java.applet.\*;

public class MyList extends Applet {

  List lista = new List("3, false);

  public void init() {

     lista.addItem("Vermelho");

     lista.addItem("Azul");

     lista.addItem("Verde");

     lista.additem("Amarelo");

     lista.addItem("Branco");

     lista.select("Branco");

     setBackground(Color.White);

     add(new Label("Escolha as cores preferidas");

     add(lista);

     add(new Button("OK");

 }

 public boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

       if((String)quem == "OK")) { // confirmado

          String cor = lista.getSelecteditens();

       if(cor == "Vermelho")

          setBackground(Color.Red);

       else if(cor == "Azul")

          setBackground(Color.Blue);

       else if(cor == "Verde")

          setBackground(Color.Green);

       else if(cor == "Amarelo")

          setBackground(Color.yellow);

       else if(cor == "Branco")

          setBackground(Color.White);

       return true;

     }

     return false;

    }

 }

}

A lista possui vários métodos para manipulação bem controlado, porém, não trataremos aqui. O interessado em usar a lista, deve procurar no livro ou manual para ter menor desperdício e maior controle na utilização de listas.

### Campo de Textos e Área de Textos

O campo de texto denominado de TextField é uma área retangular onde possa escrever ou digitar uma linha de texto e é fundamental para criar entrada/saida de dados no AWT.   
Textfield possui quatro construtores:   
public Textfield();   
public TextField(int numcols);   
public TextField(String texto);   
public TextField(String texto, int numcols);   
onde numcols é o número de colunas e texto é o texto default.   
A área de texto é análogo ao campo de texto, porém, é uma região retangular na qual permite editar texto em várias linhas. O seus construtores são:   
public TextArea();   
public Textarea(int numcols, numrows);   
public Textarea(String texto);   
public TextArea(String texto, int numcols, numrows);   
onde texto é o texto default, numcols, e numrows são número de linhas e colunas da área de texto.   
Essencialmente, a diferênça entre Textfield e TextArea , é a área de edição. Seus métodos são similares:   
Para associar um texto para ser editado, usa-se o método   
public setText(String texto);   
e o método   
public String getText();   
retorna o texto editado.   
O campo de texto ou área de texto pode ser editábel (entrada) ou não editável (saída). Para configurar a editabilidade, usa-se o método   
public void setEditable(boolean editavel);   
Se estiver usando apenas para exibir dados, basta chamar setEditable com parâmetro false. A configuração padrão de editabilidade é true.   
Como no caso de lista, TextField e TextArea dispõe de uma variedade de métodos para melhor comtrole. O interessado deve consultar o manual. Também, como na lista, o método action não é chamado. Seu evento pode ser manuseado pelo método handleEvent, ou pelo botão ou menu de confirmação. Na prática, quando deseja implementar um editor, implementamos o evento de mouse diretamente para TextArea, em vez de implementar o handleEvent().   
A seguir, um exemplo simples que usa o Textfield para entrada/saída de dados:

import java.awt.\*;

import java.applet.\*;

public class TextField extends Applet {

  String texto;

  TextField entrada(40), saida(40);

 public void init() {

     saida.setEditable(false);

     add(new Label("Entrada")); add(entrada);

     add(new Label("Saida")); add(saida);

     add(new Button("Confirma));

 }

 public boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

       if((String)quem == "Confirma")) { // confirmado

          texto = entrada.getText();

          saida.setText(texto);

          repaint(); // não esquecer

          return true;

       }

     }

     return false;

    }

 }

}

Note que, a entrada de sados no AWT é feito através de TextField e TextArea, ambos como sendo String. Quando deseja entrar ou sair om dados numéricos, devemos converter. portanto, devemos saber a conversão básica:   
(new Integer(n)).toString() converte o inteiro n para String   
(new Double(x)).toString() converte o double x para String   
(new Integer(s)).intValue() converte o String s para inteiro   
Integer. parseInt(s) converte o String s para inteiro   
(new Double(s)).doubleValue() converte o String s para double   
Double.parseDouble(s) converte o String s para double

### Scroll Bar e Canvas

O uso de Scroll Bar (barra de rolamento) requer um pouco de maturidade no AWT e não será tratado aqui. Canvas, não será necessário para iniciante em AWT e também não será tratado.

### Frames

Frame é uma janela e nele, podemos adicionar panel (dentro do panel, podemos adicionar qualquer componentes).   
A utilidade maior de um frame é criar applet que possa ser executado fora da página HTML (aplicativo) e criar janelas de comunicação. Seu construtor pode ou não receber o título da janela.   
public Frame();   
public Frame(String titulo);   
O Frame possui diversos métodos, mas veremos apenas o essencial. O interessado deve consultar o manual.   
public voidd show();   
public void hide();   
Um Frame não será mostrado quando é criado, pois pode demorar para criar, acrescentando componentes nele. Para que a janela seja visível 9e funcione), devemos chamar o método show(). Uma vez que terminou de utilizar a janela, devemos chamar o método hide() para desativar. Note que, o método hide apenas esconde a janela até que outro show() seja chamado.   
Alguns métodos úteis são:   
public void pack();   
public setResizable(boolean redimensionavel);   
onde pack() é o método que ajusta o tamanho da janela para tamanho mínimo necessário e setResizable() configura a janela para janela de tamanho ajustável ou não. Quando o parâmetro é true, é de tamanho ajustável e quando é false, é de tamanho fixo.   
Quando não vai usar mais o frame, devemos chamar o método   
public void dispose();   
para liberar o recurso que está sendo usado pelo frame. O método hide() esconde o frame, mas não libera os recursos usados.

Para criar applet que possa ser executado fora da página HTML, implementamos o método main() e através dele, criamos um Frame, adiciona a classe principal da applet e executar o init, e em seguida, executa-se o método show() do Frame. Por exemplo, se o applet chamar MeuApplet, o método main() associado nele será mais ou menos o seguinte:

import java.awt.\*;

import java.applet.\*;

public class Botao extends Applet {

  public void init(){

    add(new Button("Azul"));

    add(new Button("Vermelho"));

  }

  public void boolean action(Event evento, Object quem) {

    if(evento.target instanceof Button) {

      String label = (String)quem;

      if(label == "Azul")

         setBackground(Color.blue);

      else if (label == "Vermelho")

         setBackground(Color.red));

      else

         return false;

      repaint();

      return true; // não chama mais nenhum action seguinte

    }

    else

      return false; // evento não foi de botão. Próximo action é chamado.

  }

  public void main(String args[]) {

    Frame janela = new Frame("Applet de Botão");

    Applet principal = new Botao();

    principal.init();

    principal.start();

    janela.setLayort(new FlowLayort());

    janela.add(principal);

    janela.pack(); janela.show();

  }

}

como exemplo:

public void main(String args[]) {

  // cria a janela

  Frame janela = new Frame("Meu Applet");

  // cria o applet

  Applet principal = new MeuApplet();

  // inicializa o applet através de init() e start(): nessa ordem.

  principal.init();

  principal.start();

  // configura o Layort para Flowlayort.

  // um Frame deve ter Layout, obrigatoriamente.

  janela.setLayort(new FlowLayort());

  // Adiciona o applet na janela

  janela.add(principal);

  janela.pack(); // tamanho mínimo necessário

  janela.show(); // ativa a janela

}

### Menu